

Android 进阶解密

刘望舒◎著

电子工业出版社

Publishing House of Electronics Industry

北京•BEIJING

内 容 简 介

本书是一本 Android 进阶书籍，主要针对 Android 8.0 系统源码并结合应用开发相关知识进行介绍。本书共分为 17 章，从 3 个方面来组织内容。

第一方面介绍 Android 应用开发所需要掌握的系统源码知识，第二方面介绍 JNI、ClassLoader、Java 虚拟机、DVM&ART 虚拟机和 Hook 等技术，第三方面介绍热修复原理、插件化原理、绘制优化和内存优化等与应用开发相关的知识点。3 个方面有所关联并形成知识体系，从而使 Android 开发者能通过阅读本书达到融会贯通的目的。

本书适合有一定基础的 Android 应用开发工程师、Android 系统开发工程师和对 Android 系统源码感兴趣的读者阅读。

未经许可，不得以任何方式复制或抄袭本书之部分或全部内容。

版权所有，侵权必究。

图书在版编目（CIP）数据

Android 进阶解密 / 刘望舒著. —北京：电子工业出版社，2018.10

ISBN 978-7-121-34838-9

I. ①A… II. ①刘… III. ①移动终端—应用程序—程序设计 IV. ①TN929.53

中国版本图书馆 CIP 数据核字（2018）第 179236 号

策划编辑：付 睿

责任编辑：牛 勇 特约编辑：赵树刚

印 刷：

装 订：

出版发行：电子工业出版社

北京市海淀区万寿路 173 信箱

邮编：100036

开 本：787×980 1/16 印张：29.25 字数：702 千字

版 次：2018 年 10 月第 1 版

印 次：2018 年 10 月第 1 次印刷

定 价：99.00 元

凡所购买电子工业出版社图书有缺损问题，请向购买书店调换。若书店售缺，请与本社发行部联系，联系及邮购电话：（010）88254888，88258888。

质量投诉请发邮件至 zlts@phei.com.cn，盗版侵权举报请发邮件到 dbqq@phei.com.cn。

本书咨询联系方式：010-51260888-819，faq@phei.com.cn。

前言

为什么写这本书

Android 进阶二部曲包括《Android 进阶之光》和本书，因此写这本书的原因和《Android 进阶之光》有些关联，主要有以下几点：

（1）《Android 进阶之光》适合初、中级工程师阅读，因此我需要写一本适合中、高级工程师阅读的进阶书。

（2）目前市场上的系统源码分析的书大部分不是专门为应用开发编写的，因此我要专门为 Android 应用开发编写一本系统源码分析的书，不仅如此，我还要将系统源码和应用开发相结合。

（3）目前 Android 应用综合类进阶书籍很少，在 2017 年出版的只有《Android 进阶之光》，在 2018 年我仍要贡献出自己的力量。

（4）目前市面上的源码分析类书籍大部分是基于 Android 6.0 以前版本的，需要有一本书籍来对更新版本的系统源码进行分析。

（5）《Android 进阶之光》覆盖的知识点还远远不够，我希望能覆盖更多的知识点，让更多的人受益。

本书内容

本书共分为 17 章，各章内容如下：

- 第 1 章介绍 Android 系统架构、系统源码目录和如何阅读源码，带领大家走进 Android 系统源码的世界。

- 第 2 章介绍 Android 系统启动过程，为下面的章节做好铺垫。
- 第 3 章介绍应用程序进程启动过程。
- 第 4 章介绍四大组件的工作过程，包括根 Activity 的启动过程，Service 的启动和绑定过程，广播的注册、发送和接收过程，Content Provider 的启动过程。
- 第 5 章从源码角度分析上下文 Context。
- 第 6 章介绍 ActivityManagerService，包括 AMS 家族、AMS 的启动过程、AMS 重要的数据结构和 Activity 栈管理等内容。
- 第 7 章介绍 WindowManager，包括 WindowManager 的关联类、Window 的属性和 Window 的操作等内容。
- 第 8 章介绍 WindowManagerService，包括 WMS 的创建过程、WMS 的重要成员和 Window 的添加过程等内容。
- 第 9 章结合 MediaRecorder 框架来介绍 JNI 的原理。
- 第 10 章介绍 Android 开发所需要了解的 Java 虚拟机知识。
- 第 11 章介绍 Dalvik 和 ART 虚拟机。
- 第 12 章介绍 ClassLoader，它是理解热修复原理和插件化原理必备的知识点。
- 第 13 章介绍热修复原理，包括热修复框架的对比、资源修复、代码修复和动态链接库的修复。
- 第 14 章介绍 Hook 技术，为讲解插件化原理做铺垫。
- 第 15 章介绍插件化原理，包括插件化的产生、四大组件的插件化、资源的插件化和 so 的插件化。
- 第 16 章介绍绘制优化，包括绘制性能分析和布局优化。
- 第 17 章介绍内存优化，从避免内存泄漏开始讲起，然后介绍常用的内存分析工具 Memory Monitor、Allocation Tracker 和 Heap Dump，最后介绍分析内存泄漏的利器 MAT 和 LeakCanary。

本书特色

本书主要有以下特点：

- 本书的知识点自成体系并且环环相扣，每一个章节都或多或少地与其他章节有所关联。
- 本书是目前市面上少有的专门为 Android 应用开发者所编写的源码分析类书籍，并且将系统源码和应用开发相结合。

- 本书是目前市面上少有的讲解插件化和热修复原理的书。
- 本书为了更好地讲解知识点，会先介绍一些知识点做铺垫，比如要学习插件化原理，就需要先学习四大组件工作过程、AMS、ClassLoader 和 Hook 技术等相关知识点。

读者对象

本书适合以下读者阅读：

- 有一定基础的 Android 应用开发工程师。
- Android 系统开发工程师。
- 对 Android 系统源码感兴趣的读者。

致谢

感谢本书的策划编辑付睿，她在 CSDN 博客中发现了，并积极推动本书的出版进度，才使得本书能够及时出版。感谢我的父母以及所有关注我的朋友们，你们的鼓励和认可为我写书以及写博客带来了源源不断的动力。

勘误与互动

本人虽已竭尽全力，但书中难免会有错误，欢迎大家向我反馈，我也会在独立博客和 CSDN 博客中定期发布本书的勘误信息。

本书互动地址

独立博客：<http://liuwangshu.cn>

CSDN 博客：<http://blog.csdn.net/itachi85>

Github：<https://github.com/henrymorgen>

微信公众号：刘望舒

QQ 交流群：499174415

刘望舒

2018 年 6 月于北京

目录

第 1 章	Android 系统架构	1
1.1	Android 系统架构.....	1
1.2	Android 系统源码目录.....	4
1.2.1	整体结构	4
1.2.2	应用层部分	5
1.2.3	应用框架层部分	6
1.2.4	C/C++程序库部分	6
1.3	源码阅读	7
1.3.1	在线阅读	7
1.3.2	使用 Source Insight	9
1.4	本章小结	12
第 2 章	Android 系统启动	13
2.1	init 进程启动过程.....	13
2.1.1	引入 init 进程	13
2.1.2	init 进程的入口函数	14
2.1.3	解析 init.rc	17
2.1.4	解析 Service 类型语句	19
2.1.5	init 启动 Zygote.....	20
2.1.6	属性服务	23
2.1.7	init 进程启动总结	27

2.2	Zygote 进程启动过程	27
2.2.1	Zygote 概述	28
2.2.2	Zygote 启动脚本	28
2.2.3	Zygote 进程启动过程介绍	30
2.2.4	Zygote 进程启动总结	38
2.3	SystemServer 处理过程	39
2.3.1	Zygote 处理 SystemServer 进程	39
2.3.2	解析 SystemServer 进程	44
2.3.3	SystemServer 进程总结	48
2.4	Launcher 启动过程	48
2.4.1	Launcher 概述	48
2.4.2	Launcher 启动过程介绍	49
2.4.3	Launcher 中应用图标显示过程	54
2.5	Android 系统启动流程	59
2.6	本章小结	60
第 3 章	应用程序进程启动过程	61
3.1	应用程序进程简介	61
3.2	应用程序进程启动过程介绍	62
3.2.1	AMS 发送启动应用程序进程请求	62
3.2.2	Zygote 接收请求并创建应用程序进程	68
3.3	Binder 线程池启动过程	75
3.4	消息循环创建过程	78
3.5	本章小结	80
第 4 章	四大组件的工作过程	81
4.1	根 Activity 的启动过程	82
4.1.1	Launcher 请求 AMS 过程	82
4.1.2	AMS 到 ApplicationThread 的调用过程	85
4.1.3	ActivityThread 启动 Activity 的过程	94
4.1.4	根 Activity 启动过程中涉及的进程	99
4.2	Service 的启动过程	101

4.2.1	ContextImpl 到 AMS 的调用过程.....	101
4.2.2	ActivityThread 启动 Service.....	103
4.3	Service 的绑定过程.....	110
4.3.1	ContextImpl 到 AMS 的调用过程.....	111
4.3.2	Service 的绑定过程.....	112
4.4	广播的注册、发送和接收过程.....	122
4.4.1	广播的注册过程.....	122
4.4.2	广播的发送和接收过程.....	127
4.5	Content Provider 的启动过程.....	137
4.5.1	query 方法到 AMS 的调用过程.....	137
4.5.2	AMS 启动 Content Provider 的过程.....	143
4.6	本章小结.....	148
第 5 章	理解上下文 Context.....	149
5.1	Context 的关联类.....	149
5.2	Application Context 的创建过程.....	151
5.3	Application Context 的获取过程.....	156
5.4	Activity 的 Context 创建过程.....	156
5.5	Service 的 Context 创建过程.....	161
5.6	本章小结.....	163
第 6 章	理解 ActivityManagerService.....	164
6.1	AMS 家族.....	164
6.1.1	Android 7.0 的 AMS 家族.....	164
6.1.2	Android 8.0 的 AMS 家族.....	170
6.2	AMS 的启动过程.....	171
6.3	AMS 与应用程序进程.....	174
6.4	AMS 重要的数据结构.....	176
6.4.1	解析 ActivityRecord.....	177
6.4.2	解析 TaskRecord.....	177
6.4.3	解析 ActivityStack.....	178
6.5	Activity 栈管理.....	181

6.5.1	Activity 任务栈模型	181
6.5.2	Launch Mode	182
6.5.3	Intent 的 FLAG.....	182
6.5.4	taskAffinity	185
6.6	本章小结	186
第 7 章	理解 WindowManager	187
7.1	Window、WindowManager 和 WMS	187
7.2	WindowManager 的关联类	188
7.3	Window 的属性	193
7.3.1	Window 的类型和显示次序	193
7.3.2	Window 的标志	195
7.3.3	软键盘相关模式	196
7.4	Window 的操作	196
7.4.1	系统窗口的添加过程	197
7.4.2	Activity 的添加过程	202
7.4.3	Window 的更新过程	203
7.5	本章小结	206
第 8 章	理解 WindowManagerService	207
8.1	WMS 的职责	207
8.2	WMS 的创建过程	209
8.3	WMS 的重要成员	217
8.4	Window 的添加过程（WMS 处理部分）	219
8.5	Window 的删除过程	225
8.6	本章小结	230
第 9 章	JNI 原理	231
9.1	系统源码中的 JNI	232
9.2	MediaRecorder 框架中的 JNI.....	233
9.2.1	Java Framework 层的 MediaRecorder	233
9.2.2	JNI 层的 MediaRecorder.....	234

9.2.3 Native 方法注册	235
9.3 数据类型的转换	239
9.3.1 基本数据类型的转换	240
9.3.2 引用数据类型的转换	240
9.4 方法签名	242
9.5 解析 JNIEnv	244
9.5.1 jfieldID 和 jmethodID	245
9.5.2 使用 jfieldID 和 jmethodID	247
9.6 引用类型	249
9.6.1 本地引用	249
9.6.2 全局引用	249
9.6.3 弱全局引用	250
9.7 本章小结	251
第 10 章 Java 虚拟机	252
10.1 概述	252
10.1.1 Java 虚拟机家族	253
10.1.2 Java 虚拟机执行流程	253
10.2 Java 虚拟机结构	254
10.2.1 Class 文件格式	255
10.2.2 类的生命周期	256
10.2.3 类加载子系统	257
10.2.4 运行时数据区域	258
10.3 对象的创建	260
10.4 对象的堆内存布局	262
10.5 oop-klass 模型	263
10.6 垃圾标记算法	266
10.6.1 Java 中的引用	266
10.6.2 引用计数算法	267
10.6.3 根搜索算法	269
10.7 Java 对象在虚拟机中的生命周期	270
10.8 垃圾收集算法	271

10.8.1	标记—清除算法	271
10.8.2	复制算法	272
10.8.3	标记—压缩算法	273
10.8.4	分代收集算法	274
10.9	本章小结	275
第 11 章	Dalvik 和 ART	276
11.1	Dalvik 虚拟机	276
11.1.1	DVM 与 JVM 的区别	276
11.1.2	DVM 架构	278
11.1.3	DVM 的运行时堆	280
11.1.4	DVM 的 GC 日志	280
11.2	ART 虚拟机	281
11.2.1	ART 与 DVM 的区别	281
11.2.2	ART 的运行时堆	282
11.2.3	ART 的 GC 日志	283
11.3	DVM 和 ART 的诞生	285
11.4	本章小结	288
第 12 章	理解 ClassLoader	289
12.1	Java 中的 ClassLoader	289
12.1.1	ClassLoader 的类型	289
12.1.2	ClassLoader 的继承关系	291
12.1.3	双亲委托模式	292
12.1.4	自定义 ClassLoader	295
12.2	Android 中的 ClassLoader	298
12.2.1	ClassLoader 的类型	298
12.2.2	ClassLoader 的继承关系	300
12.2.3	ClassLoader 的加载过程	302
12.2.4	BootClassLoader 的创建	306
12.2.5	PathClassLoader 的创建	309
12.3	本章小结	311

第 13 章 热修复原理	312
13.1 热修复的产生	312
13.2 热修复框架的种类和对比	313
13.3 资源修复	314
13.3.1 Instant Run 概述	314
13.3.2 Instant Run 的资源修复	315
13.4 代码修复	318
13.4.1 类加载方案	319
13.4.2 底层替换方案	321
13.4.3 Instant Run 方案	322
13.5 动态链接库的修复	323
13.5.1 System 的 load 和 loadLibrary 方法	323
13.5.2 nativeLoad 方法分析	327
13.6 本章小结	333
第 14 章 Hook 技术	334
14.1 Hook 技术概述	334
14.2 Hook 技术分类	336
14.3 代理模式	336
14.3.1 代理模式简单实现	337
14.3.2 动态代理的简单实现	338
14.4 Hook startActivity 方法	339
14.4.1 Hook Activity 的 startActivity 方法	340
14.4.2 Hook Context 的 startActivity 方法	343
14.4.3 Hook startActivity 总结	344
14.5 本章小结	345
第 15 章 插件化原理	346
15.1 动态加载技术	346
15.2 插件化的产生	347
15.2.1 应用开发的痛点和瓶颈	347
15.2.2 插件化思想	348

15.2.3	插件化定义	350
15.3	插件化框架对比	351
15.4	Activity 插件化	352
15.4.1	Activity 的启动过程回顾	352
15.4.2	Hook IActivityManager 方案实现	354
15.4.3	Hook Instrumentation 方案实现	364
15.4.4	总结	367
15.5	Service 插件化	368
15.5.1	插件化方面 Service 与 Activity 的不同	368
15.5.2	代理分发实现	370
15.6	ContentProvider 插件化	376
15.6.1	ContentProvider 的启动过程回顾	376
15.6.2	VirtualApk 的实现	377
15.7	BroadcastReceiver 的插件化	385
15.7.1	广播插件化思路	386
15.7.2	VirtualApk 的实现	386
15.8	资源的插件化	387
15.8.1	系统资源加载	387
15.8.2	VirtualApk 实现	389
15.9	so 的插件化	390
15.10	本章小结	393
第 16 章	绘制优化	394
16.1	绘制性能分析	394
16.1.1	绘制原理	395
16.1.2	Profile GPU Rendering	396
16.1.3	Systrace	398
16.1.4	Traceview	404
16.2	布局优化	407
16.2.1	布局优化工具	407
16.2.2	布局优化方法	411
16.2.3	避免 GPU 过度绘制	419

16.3	本章小结	420
第 17 章	内存优化.....	421
17.1	避免可控的内存泄漏	421
17.1.1	什么是内存泄漏	421
17.1.2	内存泄漏的场景	422
17.2	Memory Monitor	428
17.2.1	使用 Memory Monitor.....	429
17.2.2	大内存申请与 GC.....	430
17.2.3	内存抖动	430
17.3	Allocation Tracker	430
17.3.1	使用 Allocation Tracker	431
17.3.2	alloc 文件分析	431
17.4	Heap Dump.....	434
17.4.1	使用 Heap Dump.....	434
17.4.2	检测内存泄漏	436
17.5	内存分析工具 MAT.....	438
17.5.1	生成 hprof 文件	438
17.5.2	MAT 分析 hprof 文件	440
17.6	LeakCanary	448
17.6.1	使用 LeakCanary.....	449
17.6.2	LeakCanary 应用举例	449
17.7	本章小结	453

第 1 章

Android 系统架构

本书是系统源码和应用开发相结合的书籍，想要更好地学习系统源码，就要先了解 Android 系统架构。本章将简单地介绍 Android 系统架构、系统源码目录和如何阅读源码，带领大家走进 Android 系统源码的世界，并为本书后续的章节做好知识铺垫。另外，本书作为《Android 进阶之光》的续作，本章也起到了承上启下的作用。

1.1 Android系统架构

Android 系统架构分为五层，从上到下依次是应用层、应用框架层、系统运行库层、硬件抽象层和 Linux 内核层，如图 1-1 所示。

1. 应用层（System Apps）

系统内置的应用程序以及非系统级的应用程序都属于应用层，负责与用户进行直接交互，通常都是用 Java 进行开发的。

2. 应用框架层（Java API Framework）

应用框架层为开发人员提供了开发应用程序所需要的 API，我们平常开发应用程序都是调用这一层所提供的 API，当然也包括系统应用。这一层是由 Java 代码编写的，可以称为 Java Framework。下面来看这一层所提供的主要组件，如表 1-1 所示。

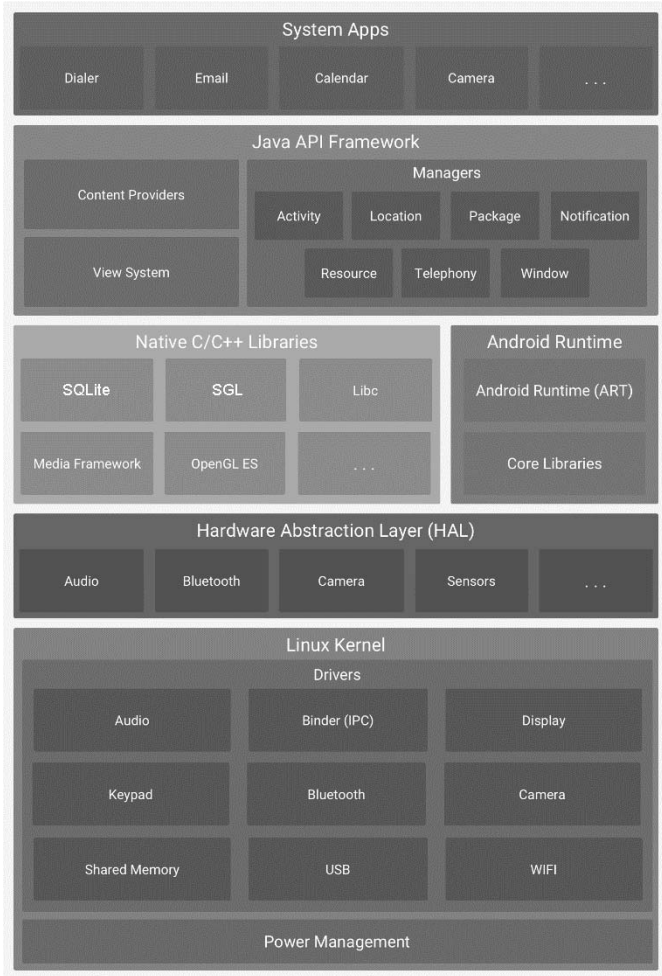


图 1-1 Android 系统架构

表 1-1 应用框架层提供的组件

名 称	功 能 描 述
Activity Manager（活动管理器）	管理各个应用程序生命周期，以及常用的导航回退功能
Location Manager（位置管理器）	提供地理位置及定位功能服务
Package Manager（包管理器）	管理所有安装在 Android 系统中的应用程序
Notification Manager（通知管理器）	使得应用程序可以在状态栏中显示自定义的提示信息
Resource Manager（资源管理器）	提供应用程序使用的各种非代码资源，如本地化字符串、图片、布局文件、颜色文件等
Telephony Manager（电话管理器）	管理所有的移动设备功能

续表

名 称	功 能 描 述
Window Manager（窗口管理器）	管理所有开启的窗口程序
Content Provider（内容提供者）	使得不同应用程序之间可以共享数据
View System（视图系统）	构建应用程序的基本组件

3. 系统运行库层（Native）

从图 1-1 可以看出，系统运行库层分为两部分，分别是 C/C++程序库和 Android 运行时库，下面分别进行介绍。

1) C/C++程序库

C/C++程序库能被 Android 系统中的不同组件所使用，并通过应用程序框架为开发者提供服务，表 1-2 列出了主要的 C/C++程序库。

表 1-2 主要的 C/C++程序库

名 称	功 能 描 述
OpenGL ES	3D 绘图函数库
Libc	从 BSD 继承来的标准 C 系统函数库，专门为基于嵌入式 Linux 的设备定制
Media Framework	多媒体库，支持多种常用的音频、视频格式录制和回放
SQLite	轻型的关系型数据库引擎
SGL	底层的 2D 图形渲染引擎
SSL	安全套接层，是一种为网络通信提供安全及数据完整性的安全协议
FreeType	可移植的字体引擎，它提供统一的接口来访问多种字体格式文件

2) Android 运行时库

从图 1-1 可以看出，运行时库又分为核心库和 ART（Android 5.0 系统之后，Dalvik 虚拟机被 ART 取代）。核心库提供了 Java 语言核心库的大多数功能，这样开发者可以使用 Java 语言来编写 Android 应用。与 JVM 相比，Dalvik 虚拟机（DVM）是专门为移动设备定制的，允许在有限的内存中同时运行多个虚拟机的实例，并且每一个 Dalvik 应用作为一个独立的 Linux 进程执行。独立的进程可以防止在虚拟机崩溃的时候所有程序都被关闭。而替代 DVM 的 ART 的机制与 DVM 不同，DVM 中的应用每次运行时，字节码都需要通过即时编译器（Just In Time，JIT）转换为机器码，这会使得应用的运行效率降低。而在 ART 中，系统在安装应用时会进行一次预编译（Ahead Of Time，AOT），将字节码预先编译成机器码并存储在本地，这样应用每次运行时就不需要执行编译了，运行效率也大大提高。

4. 硬件抽象层（HAL）

硬件抽象层是位于操作系统内核与硬件电路之间的接口层，其目的在于将硬件抽象化，为了保护硬件厂商的知识产权，它隐藏了特定平台的硬件接口细节，为操作系统提供虚拟硬件平台，使其具有硬件无关性，可在多种平台上进行移植。从软硬件测试的角度来看，软硬件的测试工作都可分别基于硬件抽象层来完成，使得软硬件测试工作的并行进行成为可能。通俗来讲，就是将控制硬件的动作放在硬件抽象层中。

5. Linux 内核层（Linux Kernel）

Android 的核心系统服务基于 Linux 内核，在此基础上添加了部分 Android 专用的驱动。系统的安全性、内存管理、进程管理、网络协议栈和驱动模型等都依赖于该内核。

了解 Android 系统的五层架构对分析系统源码有很大的帮助。

1.2 Android系统源码目录

学习Android系统源码，需要掌握系统源码目录。可以访问<http://androidxref.com>来阅读系统源码，当然，最好是将源码下载下来，下载源码的方式有很多种，这里我推荐使用百度网盘地址<http://pan.baidu.com/s/1ngsZs>进行下载，目前其中提供了Android 1.6 到Android 8.1.0 多个Android版本的源码。

1.2.1 整体结构

各个版本的源码目录基本是类似的，如果是编译后的源码目录，会多一个 out 文件夹，用来存储编译产生的文件。Android 8.0.0 的系统根目录结构说明如表 1-3 所示。

表 1-3 系统根目录结构说明

Android 源码根目录	描 述
art	全新的 ART 运行环境
bionic	系统 C 库
bootable	启动引导相关代码
build	存放系统编译规则及 generic 等基础开发包配置
cts	Android 兼容性测试套件标准
dalvik	Dalvik 虚拟机
developers	开发者目录

续表

Android 源码根目录	描 述
development	与应用程序开发相关
device	设备相关配置
docs	参考文档目录
external	开源模组相关文件
frameworks	应用程序框架，Android 系统核心部分，由 Java 和 C++编写
hardware	主要是硬件抽象层的代码
libcore	核心库相关文件
libnativehelper	动态库，实现 JNI 库的基础
out	编译完成后代码在此目录输出
pdk	Plug Development Kit 的缩写，本地开发套件
platform_testing	平台测试
prebuilts	X86 和 ARM 架构下预编译的一些资源
sdk	SDK 和模拟器
packages	应用程序包
system	底层文件系统库、应用和组件
toolchain	工具链文件
tools	工具文件
makefile	全局 Makefile 文件，用来定义编译规则

从表 1-3 可以看出，系统源码分类清晰，内容庞大且复杂，接下来分析 packages 目录中的内容，也就是应用层部分。

1.2.2 应用层部分

应用层位于整个 Android 系统的最上层，开发者开发的应用程序以及系统内置的应用程序都在应用层。源码根目录中的 packages 目录对应着系统应用层，它的目录结构如表 1-4 所示。

表 1-4 packages 目录结构

packages 目录	描 述
apps	核心应用程序
experimental	第三方应用程序
inputmethods	输入法目录
providers	内容提供者目录
screensavers	屏幕保护

续表

packages 目录	描 述
services	通信服务
wallpapers	墙纸

从目录结构可以发现，packages 目录存放着系统核心应用程序、第三方应用程序和输入法等，这些应用程序都是运行在系统应用层的，因此 packages 目录对应着系统的应用层。

1.2.3 应用框架层部分

应用框架层是系统的核心部分，一方面向上提供接口给应用层调用，另一方面向下与 C/C++程序库及硬件抽象层等进行衔接。应用框架层的主要实现代码在 frameworks/base 和 frameworks/av 目录下，其中 frameworks/base 目录结构如表 1-5 所示。

表 1-5 frameworks/base 目录

frameworks/base 目录	描 述	frameworks/base 目录	描 述
api	定义 API	cmds	重要命令：am、app_proce 等
core	核心库	data	字体和声音等数据文件
docs	文档	graphics	与图形图像相关
include	头文件	keystore	与数据签名证书相关
libs	库	location	地理位置相关库
media	多媒体相关库	native	本地库
nfc-extras	与 NFC 相关	obex	蓝牙传输
opengl	2D/3D 图形 API	packages	设置、TTS、VPN 程序
sax	XML 解析器	services	系统服务
telephony	电话通信管理	test-runner	测试工具相关
tests	与测试相关	tools	工具
vr	与 VR 相关	wifi	Wi-Fi 无线网络

1.2.4 C/C++程序库部分

系统运行库层（Native）中的 C/C++程序库的类型繁多，功能强大，C/C++程序库并不完全在一个目录中，这里给出几个常用且比较重要的 C/C++程序库所在的目录位置，如表 1-6 所示。

表 1-6 C/C++程序库所在的目录位置

目 录 位 置	描 述
bionic	Google 开发的系统 C 库，以 BSD 许可形式开源
frameworks/av/media	系统媒体库
frameworks/native/opengl	第三方图形渲染库
frameworks/native/services/surfaceflinger	图形显示库，主要负责图形的渲染、叠加和绘制等功能
external/sqlite	轻量级关系型数据库 SQLite 的 C++实现

讲完 C/C++程序库部分，其余的部分在表 1-3 中已经给出。Android 运行时库的代码在 art/目录中；硬件抽象层的代码在 hardware/目录中，这是手机厂商改动最大的部分，根据手机终端所采用的硬件平台不同会有不同的实现，剩下的目录在这里就不再详细介绍了，有兴趣的读者可以自行研究。

1.3 源码阅读

系统源码的阅读有很多种方式，总的来说分为两种：一种是在线阅读；另一种是下载源码到本地用软件工具阅读。下面分别针对这两种阅读方式进行讲解。

1.3.1 在线阅读

Android在线阅读源码的网站有很多，比如<http://www.grepcode.com>、<http://androidxref.com>、<https://www.androidos.net.cn>等，这里推荐使用<http://androidxref.com>进行在线阅读，这个网址提供了Android 1.6 到Android 8.0.0 的源码，如图 1-2 所示。

可以看到图 1-2 的左侧列出了很多 Android 版本，单击某个版本进入相应的版本界面，这里以 Android 8.0.0 版本为例，如图 1-3 所示。

左边是搜索关键字栏，右边是要搜索的源码目录列表，如果不知道要搜索的关键字具体在哪个目录中，可以单击 select all 按钮，这样会在所有的目录中进行搜索。下面以搜索 MediaPlayer.java 为例，MediaPlayer.java 在 frameworks 目录中，因此我们按照图 1-4 所示进行搜索。

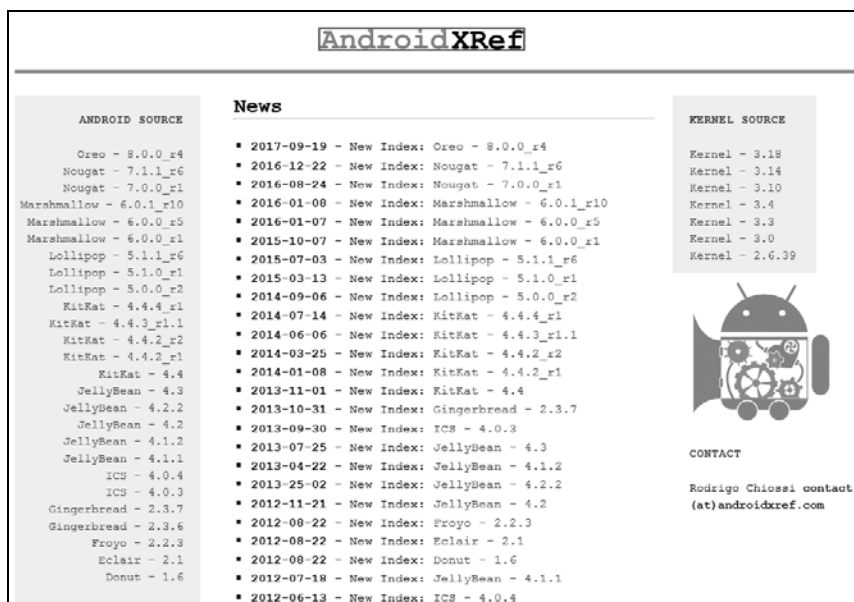


图 1-2 AndroidXRef 首页

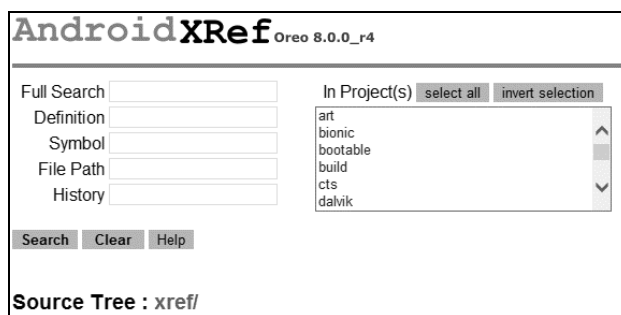


图 1-3 Android 8.0.0 版本界面

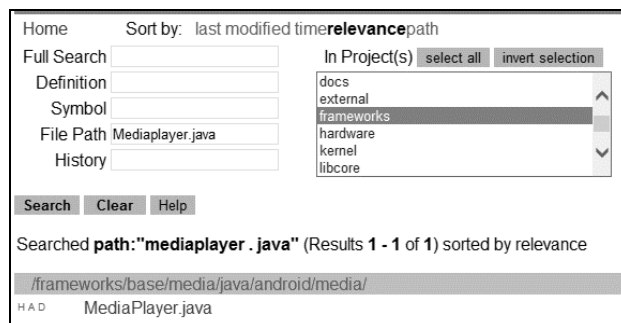


图 1-4 搜索 MediaPlayer.java

单击搜索到的 MediaPlayer.java 选项，就可以查看它的源码。如果我们想搜索 MediaPlayer.java 的 start 方法，可以在其源码界面进行搜索，也可以增加搜索关键字进行搜索，如图 1-5 所示。

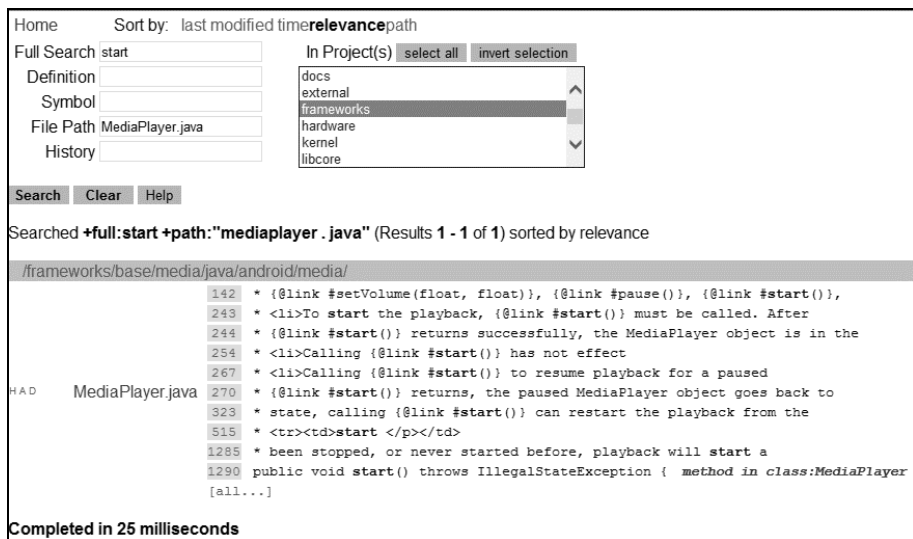


图 1-5 增加搜索关键字

1.3.2 使用Source Insight

下载源码到本地，再用软件工具阅读源码是最好的方式，这样不受网速影响，效率更高，另外在线阅读的网站都是由第三方提供的，并不是很稳定，可能某一天就访问不了了。本地阅读源码可以采用 Android Studio、Eclipse、Sublime 和 Source Insight 等软件，这里推荐使用 Source Insight。

Source Insight 是阅读源码的利器，它是 Windows 平台下的软件，很多手机开发人员都是采用 Source Insight 来阅读 Android 系统源码的。

1. 新建源码项目

安装 Source Insight 软件后，首先要新建源码项目。通过选择菜单项 Project→New Project，会弹出如图 1-6 所示的对话框。

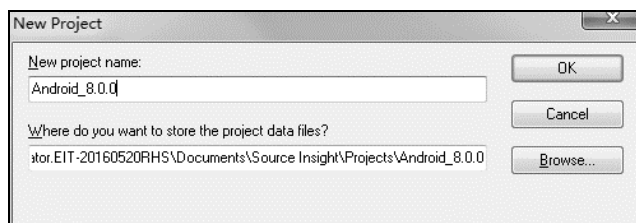


图 1-6 新建源码项目

这里我们指定源码项目的名称为 Android_8.0.0，然后单击 OK 按钮打开 New Project Settings 对话框，如图 1-7 所示。

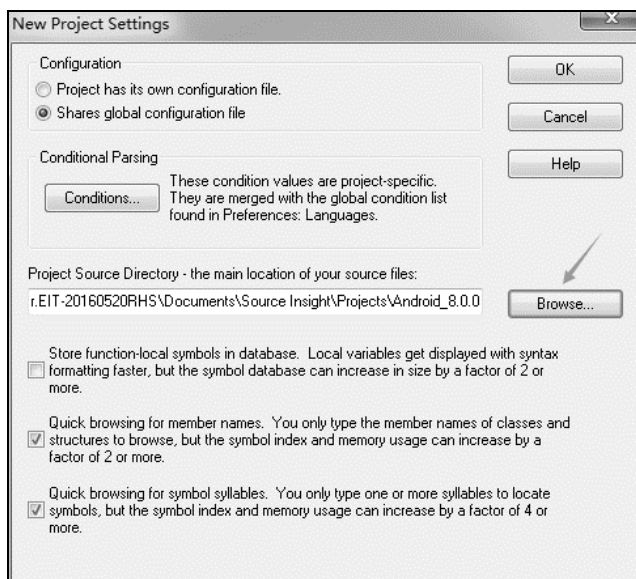


图 1-7 New Project Settings 对话框

单击图 1-7 箭头指向的 Browse 按钮来选择本地系统源码所在的路径，比如我的系统源码路径为 D:/Android/android-8.0.0_r1。选择好加载路径后单击 OK 按钮会打开 Add and Remove Project Files 对话框，在这个对话框中可以向项目中添加整个 Android 系统源码，也可以只把源码部分目录添加到项目中，以后再根据需要添加其他目录。如果向项目添加整个 Android 系统源码，加载时会非常慢，这里我们只添加本书相关的源码目录 Frameworks、Libcore、Packages、System、Art 和 Libnativehelper，这几个目录基本上可以满足日常的系统源码阅读需求了，如图 1-8 所示。

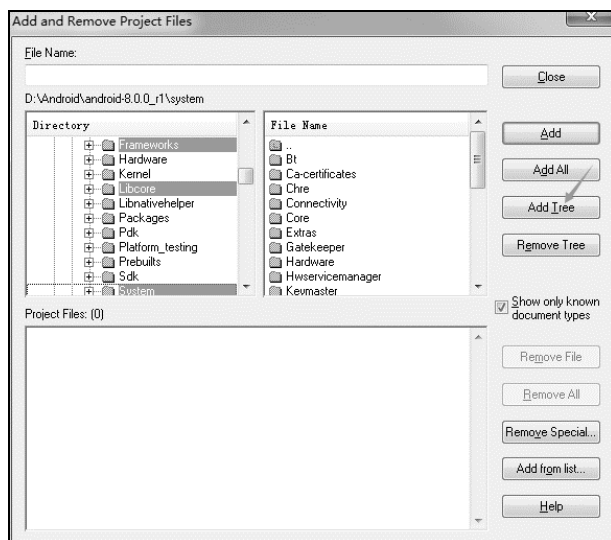


图 1-8 Add and Remove Project Files 对话框

单击图 1-8 中箭头指向的 Add Tree 按钮，就会将选择的目录源码加载到 Android_8.0.0 项目中，这个时候会弹出加载进度条，加载完毕后单击对话框的“关闭”按钮就可以了。

2. 定位文件

Source Insight 的定位文件功能十分强大，我们只需要知道源码文件名就可以轻松找到它，比如我们要找 MediaPlayer.java，只要在文件搜索框中输入 MediaPlayer.java 即可，如图 1-9 所示。

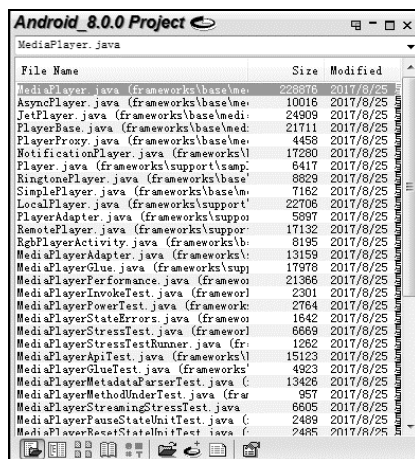


图 1-9 定位文件

单击搜索到的 MediaPlayer.java 选项, 就可以打开展示 MediaPlayer.java 源码的对话框。

3. 全局搜索

Source Insight 另一个好用的功能就是全局搜索, 默认快捷键为 Ctrl+/, 或者单击工具栏中类似 R 的图标, 弹出的搜索对话框如图 1-10 所示。在 Search In 下拉列表框中可以自定义搜索的范围, 比如我们想查找所有 Java 文件中引用 MediaPlayer 类的情况, 就可以按如图 1-10 所示进行操作。

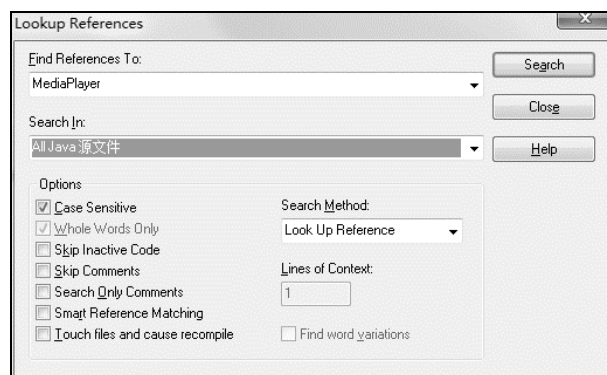


图 1-10 全局搜索

当然, Source Insight 的功能远不止以上几种, 相信随着使用次数的增多, 你会熟练掌握它的大部分功能, 这里就不过多介绍了。

1.4 本章小结

本章的内容比较少, 主要介绍了 Android 系统架构、系统源码目录和如何阅读源码, 并且将系统架构和系统源码进行了关联。关于如何阅读源码这里介绍了两种方法, 分别是在线阅读和使用 Source Insight 阅读。本章所讲的内容都是深入源码学习的必备知识点, 也为本书后续的章节做好了知识铺垫。

第 2 章

Android 系统启动

作为本书的第 2 章，可能你会觉得诧异，为何在书的开始就要介绍 Android 系统启动呢？这里有必要说明一下，Android 系统启动与本书后续很多内容都有关联，比如应用进程启动流程、四大组件原理、AMS、ClassLoader 等，而 ClassLoader 又是热修复和插件化的基础，可见 Android 系统启动是十分重要并且需要首先学习的知识点，这里也建议大家先阅读完本章的内容后再去阅读后面的章节。本章简要介绍 Android 系统启动的流程，不会拘泥于源码细节，旨在让读者了解大概的流程。另外，需要提醒大家注意的一点是，阅读本章需要有一定的 C/C++ 基础。

2.1 init 进程启动过程

init 进程是 Android 系统中用户空间的第一个进程，进程号为 1，是 Android 系统启动流程中一个关键的步骤，作为第一个进程，它被赋予了很多极其重要的工作职责，比如创建 Zygote（孵化器）和属性服务等。init 进程是由多个源文件共同组成的，这些文件位于源码目录 system/core/init 中。

2.1.1 引入 init 进程

为了讲解 init 进程，首先要了解 Android 系统启动流程的前几步，以引入 init 进程。

1. 启动电源以及系统启动

当电源按下时引导芯片代码从预定义的地方（固化在 ROM）开始执行。加载引导程序 BootLoader 到 RAM 中，然后执行。

2. 引导程序 BootLoader

引导程序 BootLoader 是在 Android 操作系统开始运行前的小程序，它的主要作用是把系统 OS 拉起来并运行。

3. Linux 内核启动

当内核启动时，设置缓存、被保护存储器、计划列表、加载驱动。在内核完成系统设置后，它首先在系统文件中寻找 init.rc 文件，并启动 init 进程。

4. init 进程启动

init 进程做的工作比较多，主要用来初始化和启动属性服务，也用来启动 Zygote 进程。

从上面的步骤可以看出，当我们按下启动电源时，系统启动后会加载引导程序，引导程序又启动 Linux 内核，在 Linux 内核加载完成后，第一件事就是要启动 init 进程。关于 Android 系统启动的完整流程会在本章的 2.5 节进行讲解，这一节的任务就是先了解 init 进程的启动过程。

2.1.2 init进程的入口函数

在 Linux 内核加载完成后，它首先在系统文件中寻找 init.rc 文件，并启动 init 进程，然后查看 init 进程的入口函数 main，代码如下所示：

```
system/core/init/init.cpp
```

```
int main(int argc, char** argv) {
    if (!strcmp(basename(argv[0]), "ueventd")) {
        return ueventd_main(argc, argv);
    }
    if (!strcmp(basename(argv[0]), "watchdogd")) {
        return watchdogd_main(argc, argv);
    }
    if (REBOOT_BOOTLOADER_ON_PANIC) {
        install_reboot_signal_handlers();
    }
    add_environment("PATH", _PATH_DEFPATH);
}
```

```

bool is_first_stage = (getenv("INIT_SECOND_STAGE") == nullptr);
if (is_first_stage) {
    boot_clock::time_point start_time = boot_clock::now();
    //清理 umask
    umask(0);
    //创建和挂载启动所需的文件目录
    mount("tmpfs", "/dev", "tmpfs", MS_NOSUID, "mode=0755");
    mkdir("/dev/pts", 0755);
    mkdir("/dev/socket", 0755);
    mount("devpts", "/dev/pts", "devpts", 0, NULL);
    #define MAKE_STR(x) __STRING(x)
    mount("proc", "/proc", "proc", 0, "hidepid=2,gid=" MAKE_STR(AID_READPROC));
    chmod("/proc/cmdline", 0440);
    gid_t groups[] = { AID_READPROC };
    setgroups(arraysize(groups), groups);
    mount("sysfs", "/sys", "sysfs", 0, NULL);
    mount("selinuxfs", "/sys/fs/selinux", "selinuxfs", 0, NULL);
    mknod("/dev/kmsg", S_IFCHR | 0600, makedev(1, 11));
    mknod("/dev/random", S_IFCHR | 0666, makedev(1, 8));
    mknod("/dev/urandom", S_IFCHR | 0666, makedev(1, 9));
    //初始化 Kernel 的 Log, 这样就可以从外界获取 Kernel 的日志
    InitKernelLogging(argv);
    ...
}

...
//对属性服务进行初始化
property_init();//1
...
//创建 epoll 句柄
epoll_fd = epoll_create1(EPOCH_CLOEXEC);
if (epoll_fd == -1) {
    PLOG(ERROR) << "epoll_create1 failed";
    exit(1);
}
//用于设置子进程信号处理函数, 如果子进程 (Zygote 进程) 异常退出, init 进程会调用该函数中设定
//的信号处理函数来进行处理
signal_handler_init();//2
//导入默认的环境变量
property_load_boot_defaults();
export_oem_lock_status();
//启动属性服务
start_property_service();//3
set_usb_controller();

```

```

const BuiltinFunctionMap function_map;
Action::set_function_map(&function_map);
Parser& parser = Parser::GetInstance();
parser.AddSectionParser("service", std::make_unique<ServiceParser>());
parser.AddSectionParser("on", std::make_unique<ActionParser>());
parser.AddSectionParser("import", std::make_unique<ImportParser>());
std::string bootscript = GetProperty("ro.boot.init_rc", "");
if (bootscript.empty()) {
    //解析 init.rc 配置文件
    parser.ParseConfig("/init.rc");//4
    parser.set_is_system_etc_init_loaded(
        parser.ParseConfig("/system/etc/init"));
    parser.set_is_vendor_etc_init_loaded(
        parser.ParseConfig("/vendor/etc/init"));
    parser.set_is_odm_etc_init_loaded(parser.ParseConfig("/odm/etc/init"));
} else {
    parser.ParseConfig(bootscript);
    parser.set_is_system_etc_init_loaded(true);
    parser.set_is_vendor_etc_init_loaded(true);
    parser.set_is_odm_etc_init_loaded(true);
}
if (false) parser.DumpState();
ActionManager& am = ActionManager::GetInstance();
...
while (true) {
    int epoll_timeout_ms = -1;
    if (!(waiting_for_prop || ServiceManager::GetInstance().IsWaitingForExec())) {
        //内部遍历执行每个 action 中携带的 command 对应的执行函数
        am.ExecuteOneCommand();
    }
    if (!(waiting_for_prop || ServiceManager::GetInstance().IsWaitingForExec())) {
        //重启死去的进程
        restart_processes();//5
        if (process_needs_restart_at != 0) {
            epoll_timeout_ms = (process_needs_restart_at - time(nullptr)) * 1000;
            if (epoll_timeout_ms < 0) epoll_timeout_ms = 0;
        }
        if (am.HasMoreCommands()) epoll_timeout_ms = 0;
    }
    epoll_event ev;
    int nr = TEMP_FAILURE_RETRY(epoll_wait(epoll_fd, &ev, 1, epoll_timeout_ms));
    if (nr == -1) {
        PLOG(ERROR) << "epoll_wait failed";
    } else if (nr == 1) {

```

```

        ((void (*)( )) ev.data.ptr)();
    }
}
return 0;
}

```

init 的 main 函数做了很多事情，比较复杂，我们只需关注主要的几点就可以了。在开始的时候创建和挂载启动所需的文件目录，其中挂载了 tmpfs、devpts、proc、sysfs 和 selinuxfs 共 5 种文件系统，这些都是系统运行时目录，顾名思义，只在系统运行时才会存在，系统停止时会消失。

在注释 1 处调用 property_init 函数来对属性进行初始化，并在注释 3 处调用 start_property_service 函数启动属性服务，关于属性服务，后面会讲到。在注释 2 处调用 signal_handler_init 函数用于设置子进程信号处理函数，它被定义在 system/core/init/signal_handler.cpp 中，主要用于防止 init 进程的子进程成为僵尸进程，为了防止僵尸进程的出现，系统会在子进程暂停和终止的时候发出 SIGCHLD 信号，而 signal_handler_init 函数就是用来接收 SIGCHLD 信号的（其内部只处理进程终止的 SIGCHLD 信号）。

假设 init 进程的子进程 Zygote 终止了，signal_handler_init 函数内部会调用 handle_signal 函数，经过层层函数调用和处理，最终会找到 Zygote 进程并移除所有的 Zygote 进程的信息，再重启 Zygote 服务的启动脚本（比如 init.zygote64.rc）中带有 onrestart 选项的服务，关于 init.zygote64.rc 后面会讲到，至于 Zygote 进程本身会在注释 5 处被重启。这里只是拿 Zygote 进程举个例子，其他 init 进程子进程的原理也是类似的。

注释 4 处用来解析 init.rc 文件，解析 init.rc 的文件为 system/core/init/init_parse.cpp 文件，接下来我们查看 init.rc 里做了什么。

僵尸进程与危害

在 UNIX/Linux 中，父进程使用 fork 创建子进程，在子进程终止之后，如果父进程并不知道子进程已经终止了，这时子进程虽然已经退出了，但是在系统进程表中还为它保留了一定的信息（比如进程号、退出状态、运行时间等），这个子进程就被称作僵尸进程。系统进程表是一项有限资源，如果系统进程表被僵尸进程耗尽的话，系统就可能无法创建新的进程了。

2.1.3 解析 init.rc

init.rc 是一个非常重要的配置文件，它是由 Android 初始化语言(Android Init Language)

编写的脚本,这种语言主要包含 5 种类型语句: Action、Command、Service、Option 和 Import。
init.rc 的配置代码如下所示:

```
system/core/rootdir/init.rc

on init
    sysclktz 0
    copy /proc/cmdline /dev/urandom
    copy /default.prop /dev/urandom
    ...
on boot
    ifup lo
    hostname localhost
    domainname localdomain
    setrlimit 13 40 40
    ...
```

这里只截取了一部分代码。on init 和 on boot 是 Action 类型语句,它的格式如下所示:

```
on <trigger> [&& <trigger>]*    //设置触发器
    <command>
    <command>    //动作触发之后要执行的命令
```

为了分析如何创建 Zygote,我们主要查看 Service 类型语句,它的格式如下所示:

```
service <name> <pathname> [ <argument> ]*    //<service 的名字><执行程序路径><传递参数>
    <option>    //option 是 service 的修饰词,影响什么时候、如何启动 Service
    <option>
    ...
```

需要注意的是,在 Android 8.0 中对 init.rc 文件进行了拆分,每个服务对应一个 rc 文件。我们要分析的 Zygote 启动脚本则在 init.zygoteXX.rc 中定义,这里拿 64 位处理器为例,init.zygote64.rc 的代码如下所示:

```
system/core/rootdir/init.zygote64.rc

service zygote /system/bin/app_process64 -Xzygote /system/bin --zygote --start-
system-server
    class main
    priority -20
    user root
    group root readproc
    socket zygote stream 660 root system
    onrestart write /sys/android_power/request_state wake
    onrestart write /sys/power/state on
    onrestart restart audioserver
```



```
onrestart restart cameraserver
onrestart restart media
onrestart restart netd
onrestart restart wificond
writepid /dev/cpuset/foreground/tasks
```

根据 Service 类型语句的格式我们来大概分析上面代码的意思。Service 用于通知 init 进程创建名为 zygote 的进程，这个进程执行程序的路径为/system/bin/app_process64^①，其后面的代码是要传给 app_process64 的参数。class main 指的是 Zygote 的 classname 为 main^②，后面会用到它。关于 Zygote 启动脚本会在本章的 2.2.2 节进行详细介绍。（此处标注的①、②，后续内容会引用到。）

2.1.4 解析Service类型语句

init.rc 中的 Action 类型语句和 Service 类型语句都有相应的类来进行解析，Action 类型语句采用 ActionParser 来进行解析，Service 类型语句采用 ServiceParser 来进行解析，这里因为主要分析 Zygote，所以只介绍 ServiceParser。ServiceParser 的实现代码在 system/core/init/service.cpp 中，接下来我们来查看 ServiceParser 是如何解析上面提到的 Service 类型语句的，会用到两个函数：一个是 ParseSection，它会解析 Service 的 rc 文件，比如上文讲到的 init.zygote64.rc，ParseSection 函数主要用来搭建 Service 的架子；另一个是 ParseLineSection，用于解析子项。代码如下所示：

system/core/init/service.cpp

```
bool ServiceParser::ParseSection(const std::vector<std::string>& args,
                                std::string* err) {
    if (args.size() < 3) { //判断 Service 是否有 name 与可执行程序
        *err = "services must have a name and a program";
        return false;
    }

    const std::string& name = args[1];
    if (!IsValidName(name)) { //检查 Service 的 name 是否有效
        *err = StringPrintf("invalid service name '%s'", name.c_str());
        return false;
    }

    std::vector<std::string> str_args(args.begin() + 2, args.end());
    service_ = std::make_unique<Service>(name, str_args); //1
    return true;
}
```

```
bool ServiceParser::ParseLineSection(const std::vector<std::string>& args,
                                     const std::string& filename, int line,
                                     std::string* err) const {
    return service_ ? service_>ParseLine(args, err) : false;
}
```

注释 1 处, 根据参数, 构造出一个 Service 对象, 它的 classname 为 default。在解析完所有数据后, 会调用 EndSection 函数:

system/core/init/service.cpp

```
void ServiceParser::EndSection() {
    if (service_) {
        ServiceManager::GetInstance().AddService(std::move(service_));
    }
}
```

EndSection 函数中会调用 ServiceManager 的 AddService 函数, 接着查看 AddService 函数做了什么:

system/core/init/service.cpp

```
void ServiceManager::AddService(std::unique_ptr<Service> service) {
    Service* old_service = FindServiceByName(service->name());
    if (old_service) {
        LOG(ERROR) << "ignored duplicate definition of service '" << service->name()
        << "'";
        return;
    }
    services_.emplace_back(std::move(service)); //1
}
```

注释 1 处的代码将 Service 对象加入 Service 链表中。上面的 Service 解析过程总体来讲就是根据参数创建出 Service 对象, 然后根据选项域的内容填充 Service 对象, 最后将 Service 对象加入 vector 类型的 Service 链表中。

2.1.5 init启动Zygote

讲完了解析 Service, 接下来该讲 init 是如何启动 Service 的, 在这里主要讲解启动 Zygote 这个 Service。在 Zygote 的启动脚本中, 我们可知 Zygote 的 classname 为 main。在 init.rc 中有如下配置代码:

system/core/rootdir/init.rc

```
...
on nonencrypted
    exec - root -- /system/bin/update_verifier nonencrypted
    class_start main //1
    class_start late_start
...
```

其中 class_start 是一个 COMMAND，对应的函数为 do_class_start。注释 1 处启动那些 classname 为 main 的 Service，从 2.1.3 节末段的标注②处，我们知道 Zygote 的 classname 就是 main，因此 class_start main 是用来启动 Zygote 的。do_class_start 函数在 builtins.cpp 中定义，如下所示：

system/core/init/builtins.cpp

```
static int do_class_start(const std::vector<std::string>& args) {
    ServiceManager::GetInstance().
        ForEachServiceInClass(args[1], [] (Service* s) { s->StartIfNotDisabled(); });
    return 0;
}
```

ForEachServiceInClass 函数会遍历 Service 链表，找到 classname 为 main 的 Zygote，并执行 StartIfNotDisabled 函数，如下所示：

system/core/init/service.cpp

```
bool Service::StartIfNotDisabled() {
    if (!(flags_ & SVC_DISABLED)) { //1
        return Start();
    } else {
        flags_ |= SVC_DISABLED_START;
    }
    return true;
}
```

注释 1 处，如果 Service 没有在其对应的 rc 文件中设置 disabled 选项，则会调用 Start 函数启动该 Service，Zygote 对应的 init.zygote64.rc 中并没有设置 disabled 选项，因此我们接着来查看 Start 函数，如下所示：

system/core/init/service.cpp

```
bool Service::Start() {
    flags_ &= ~(SVC_DISABLED|SVC_RESTARTING|SVC_RESET|SVC_RESTART|SVC_DISABLED_
        START));
    time_started_ = 0;
```

```

//如果 Service 已经运行, 则不启动
if (flags_ & SVC_RUNNING) {
    return false;
}
bool needs_console = (flags_ & SVC_CONSOLE);
if (needs_console && !have_console) {
    ERROR("service '%s' requires console\n", name_.c_str());
    flags_ |= SVC_DISABLED;
    return false;
}
//判断需要启动的 Service 的对应的执行文件是否存在, 不存在则不启动该 Service
struct stat sb;
if (stat(args_[0].c_str(), &sb) == -1) {
    ERROR("cannot find '%s' (%s), disabling '%s'\n",
        args_[0].c_str(), strerror(errno), name_.c_str());
    flags_ |= SVC_DISABLED;
    return false;
}
...
//如果子进程没有启动, 则调用 fork 函数创建子进程
pid_t pid = fork();//1
//当前代码逻辑在子进程中运行
if (pid == 0) {//2
    umask(077);
    ...
    //调用 execve 函数, Service 子进程就会被启动
    if (execve(args_[0].c_str(), (char**) &strs[0], (char**) ENV) < 0) {//3
        ERROR("cannotexecve('%s'): %s\n", args_[0].c_str(), strerror(errno)); }
        _exit(127);
    }
    ...
    return true;
}

```

首先判断 Service 是否已经运行, 如果运行则不再启动, 直接返回 false。如果程序走到注释 1 处, 说明子进程还没有被启动, 就调用 fork 函数创建子进程, 并返回 pid 值, 注释 2 处如果 pid 值为 0, 则说明当前代码逻辑在子进程中运行。注释 3 处在子进程中调用 execve 函数, Service 子进程就会被启动, 并进入该 Service 的 main 函数中, 如果该 Service 是 Zygote, 从 2.1.3 节末段的标注①处我们可知 Zygote 执行程序的路径为/system/bin/app_process64, 对应的文件为 app_main.cpp, 这样就会进入 app_main.cpp 的 main 函数中, 也就是在 Zygote 的 main 函数中, 代码如下:

```
frameworks/base/cmds/app_process/app_main.cpp
```

```
int main(int argc, char* const argv[])
{
    ...
    if (zygote) {
        runtime.start("com.android.internal.os.ZygoteInit", args, zygote); //1
    } else if (className) {
        runtime.start("com.android.internal.os.RuntimeInit", args, zygote);
    } else {
        fprintf(stderr, "Error: no class name or --zygote supplied.\n");
        app_usage();
        LOG_ALWAYS_FATAL("app_process: no class name or --zygote supplied.");
        return 10;
    }
}
```

从注释 1 处的代码可以得知调用 runtime 的 start 函数启动 Zygote，至此 Zygote 就启动了。

2.1.6 属性服务

Windows 平台上有一个注册表管理器，注册表的内容采用键值对的形式来记录用户、软件的一些使用信息。即使系统或者软件重启，其还是能够根据之前注册表中的记录，进行相应的初始化工作。Android 也提供了一个类似的机制，叫作属性服务。

init 进程启动时会启动属性服务，并为其分配内存，用来存储这些属性，如果需要这些属性直接读取就可以了，在 2.1.2 节的开头部分，我们提到在 init.cpp 的 main 函数中与属性服务相关的代码有以下两行：

```
system/core/init/init.cpp
```

```
property_init();
start_property_service();
```

这两行代码用来初始化属性服务配置并启动属性服务。首先我们来学习属性服务配置的初始化和启动。

1. 属性服务初始化与启动

property_init 函数的具体实现代码如下所示：

```
system/core/init/property_service.cpp
```

```
void property_init() {
    if (__system_property_area_init()) {
        LOG(ERROR) << "Failed to initialize property area";
        exit(1);
    }
}
```

__system_property_area_init 函数用来初始化属性内存区域。接下来查看 start_property_service 函数的具体代码：

```
system/core/init/property_service.cpp
```

```
void start_property_service() {
    property_set("ro.property_service.version", "2");
    property_set_fd = create_socket(PROP_SERVICE_NAME, SOCK_STREAM | SOCK_CLOEXEC
    | SOCK_NONBLOCK, 0666, 0, 0, NULL);//1

    if (property_set_fd == -1) {
        PLOG(ERROR) << "start_property_service socket creation failed";
        exit(1);
    }
    listen(property_set_fd, 8);//2
    register_epoll_handler(property_set_fd, handle_property_set_fd);//3
}
```

在注释 1 处创建非阻塞的 Socket。在注释 2 处调用 listen 函数对 property_set_fd 进行监听，这样创建的 Socket 就成为 server，也就是属性服务；listen 函数的第二个参数设置为 8，意味着属性服务最多可以同时为 8 个试图设置属性的用户提供服务。注释 3 处的代码将 property_set_fd 放入了 epoll 中，用 epoll 来监听 property_set_fd：当 property_set_fd 中有数据到来时，init 进程将调用 handle_property_set_fd 函数进行处理。

在 Linux 新的内核中，epoll 用来替换 select，epoll 是 Linux 内核为处理大批量文件描述符而做了改进的 poll，是 Linux 下多路复用 I/O 接口 select/poll 的增强版本，它能显著提高程序在大量并发连接中只有少量活跃的情况下的系统 CPU 利用率。epoll 内部用于保存事件的数据类型是红黑树，查找速度快，select 采用的数组保存信息，查找速度很慢，只有当等待少量文件描述符时，epoll 和 select 的效率才会差不多。

2. 服务处理客户端请求

从上面我们得知，属性服务接收到客户端的请求时，会调用 handle_property_set_fd 函数进行处理：

```
system/core/init/property_service.cpp
```

```
static void handle_property_set_fd() {
...
    switch (cmd) {
    case PROP_MSG_SETPROP: {
        char prop_name[PROP_NAME_MAX];
        char prop_value[PROP_VALUE_MAX];
        //如果 Socket 读取不到属性数据则返回
        if (!socket.RecvChars(prop_name, PROP_NAME_MAX, &timeout_ms) ||
            !socket.RecvChars(prop_value, PROP_VALUE_MAX, &timeout_ms)) {
            PLOG(ERROR) << "sys_prop(PROP_MSG_SETPROP): error while reading name/value
            from the socket";
            return;
        }
        prop_name[PROP_NAME_MAX-1] = 0;
        prop_value[PROP_VALUE_MAX-1] = 0;
        handle_property_set(socket, prop_value, prop_value, true);//1
        break;
    }
    ...
    }
}
```

Android 7.0 中只用 `handle_property_set_fd` 函数来处理客户端请求，Android 8.0 的源码中则增加了注释 1 处的 `handle_property_set` 函数做进一步封装处理，如下所示：

```
system/core/init/property_service.cpp
```

```
static void handle_property_set(SocketConnection& socket,
                                const std::string& name,
                                const std::string& value,
                                bool legacy_protocol) {
...
    //如果属性名称以“ctl.”开头，说明是控制属性
    if (android::base::StartsWith(name, "ctl.")) {//1
        //检查客户端权限
        if (check_control_mac_perms(value.c_str(), source_ctx, &cr)) {
            //设置控制属性
            handle_control_message(name.c_str() + 4, value.c_str());//2
            if (!legacy_protocol) {
                socket.SendUint32(PROP_SUCCESS);
            }
        } else {
            ...
        }
    }
```

```

    } else {
        //该分支是普通属性
        //检查客户端权限
        if (check_mac_perms(name, source_ctx, &cr)) {
            uint32_t result = property_set(name, value); //3
            if (!legacy_protocol) {
                socket.SendUint32(result);
            }
        } else {
            LOG(ERROR) << "sys_prop(" << cmd_name << "): permission denied uid:" << cr.uid
            << " name:" << name;
            if (!legacy_protocol) {
                socket.SendUint32(PROP_ERROR_PERMISSION_DENIED);
            }
        }
    }
    freecon(source_ctx);
}

```

系统属性分为两种类型：一种是普通属性；还有一种是控制属性，控制属性用来执行一些命令，比如开机的动画就使用了这种属性。因此，handle_property_set 函数分为了两个处理分支，一部分处理控制属性，另一部分用于处理普通属性，这里只分析处理普通属性。如果注释 1 处的属性名称以“ctl.”开头，就说明是控制属性，如果客户端权限满足，则会调用 handle_control_message 函数来修改控制属性。如果是普通属性，则会在客户端权限满足的条件下调用注释 3 处的 property_set 函数来对普通属性进行修改，如下所示：

system/core/init/property_service.cpp

```

uint32_t property_set(const std::string& name, const std::string& value) {
    size_t valuelen = value.size();
    //判断属性是否合法
    if (!is_legal_property_name(name)) {
        LOG(ERROR) << "property_set(\"" << name << "\", \"" << value << "\") failed:
        bad name";
        return PROP_ERROR_INVALID_NAME;
    }
    ...
    //从属性存储空间查找该属性
    prop_info* pi = (prop_info*) __system_property_find(name.c_str()); //1
    //如果属性存在
    if (pi != nullptr) {
        //如果属性名称以“ro.”开头，则表示只读，不能修改，直接返回
        if (android::base::StartsWith(name, "ro.")) {

```



```

        LOG(ERROR) << "property_set(\"" << name << "\", \"" << value << "\") failed: "
            << "property already set";
        return PROP_ERROR_READ_ONLY_PROPERTY;
    }
    //如果属性存在,就更新属性值
    __system_property_update(pi, value.c_str(), valuelen);
} else {
    //如果属性不存在,则添加该属性
    int rc = __system_property_add(name.c_str(), name.size(), value.c_str(),
        valuelen);
    if (rc < 0) {
        LOG(ERROR) << "property_set(\"" << name << "\", \"" << value << "\") failed: "
            << "__system_property_add failed";
        return PROP_ERROR_SET_FAILED;
    }
}
//属性名称以“persist.”开头的处理部分
if (persistent_properties_loaded && android::base::StartsWith(name, "persist.")) {
    write_persistent_property(name.c_str(), value.c_str());
}
property_changed(name, value);
return PROP_SUCCESS;
}

```

property_set 函数主要对普通属性进行修改,首先要判断该属性是否合法,如果合法就在注释 1 处从属性存储控件中查找该属性,如果属性存在,就更新属性值,否则就添加该属性。另外,还对名称以“ro”“persist”开头的属性进行了相应的处理。

2.1.7 init进程启动总结

init 进程启动做了很多的工作,总的来说主要做了以下三件事:

- (1) 创建和挂载启动所需的文件目录。
- (2) 初始化和启动属性服务。
- (3) 解析 init.rc 配置文件并启动 Zygote 进程。

2.2 Zygote进程启动过程

在 2.1 节中我们学习了 init 进程启动过程,在启动过程中主要做了三件事,其中一件就

是创建了 Zygote 进程，本节接着学习 Zygote 进程启动过程，首先我们要了解 Zygote 是什么。

2.2.1 Zygote概述

在 Android 系统中，DVM（Dalvik 虚拟机）和 ART、应用程序进程以及运行系统的关键服务的 SystemServer 进程都是由 Zygote 进程来创建的，我们也将它称为孵化器。它通过 fork（复制进程）的形式来创建应用程序进程和 SystemServer 进程，由于 Zygote 进程在启动时会创建 DVM 或者 ART，因此通过 fork 而创建的应用程序进程和 SystemServer 进程可以在内部获取一个 DVM 或者 ART 的实例副本。

我们已经知道 Zygote 进程是在 init 进程启动时创建的，起初 Zygote 进程的名称并不是叫“zygote”，而是叫“app_process”，这个名称是在 Android.mk 中定义的，Zygote 进程启动后，Linux 系统下的 pctrl 系统会调用 app_process，将其名称换成了“zygote”。

2.2.2 Zygote启动脚本

在 init.rc 文件中采用了 Import 类型语句来引入 Zygote 启动脚本，这些启动脚本都是由 Android 初始化语言（Android Init Language）来编写的：

```
import /init.${ro.zygote}.rc
```

可以看出 init.rc 不会直接引入一个固定的文件，而是根据属性 ro.zygote 的内容来引入不同的文件。

从 Android 5.0 开始，Android 开始支持 64 位程序，Zygote 也就有了 32 位和 64 位的区别，所以在这里用 ro.zygote 属性来控制使用不同的 Zygote 启动脚本，从而也就启动了不同版本的 Zygote 进程，ro.zygote 属性的取值有以下 4 种：

- init.zygote32.rc
- init.zygote32_64.rc
- init.zygote64.rc
- init.zygote64_32.rc

这些 Zygote 启动脚本都放在 system/core/rootdir 目录中，为了更好地分析这些 Zygote 启动脚本，我们先来回顾一下 2.1.3 节所提到的 Android 初始化语言的 Service 类型语句，它的格式如下所示：

```

service <name> <pathname> [ <argument> ]*    //<service 的名字><执行程序路径><传递参数>
    <option>          //option 是 Service 的修饰词，影响什么时候、如何启动 Service
    <option>
    ...

```

了解了 Service 类型语句的格式，下面分别介绍这些 Zygote 启动脚本。

1. init.zygote32.rc

表示支持纯 32 位程序，init.zygote32.rc 文件内容如下所示：

```

service zygote /system/bin/app_process -Xzygote /system/bin --zygote --start-
system-server
    class main
    priority -20
    user root
    group root readproc
    socket zygote stream 660 root system
    onrestart write /sys/android_power/request_state wake
    onrestart write /sys/power/state on
    onrestart restart audioserver
    onrestart restart camerasetter
    onrestart restart media
    onrestart restart netd
    onrestart restart wificond
    writepid /dev/cpuset/foreground/tasks

```

根据 Service 类型语句的格式，可以得知 Zygote 进程名称为 zygote，执行程序为 app_process，classname 为 main，如果 audioserver、cameraserver、media 等进程终止了，就需要进行 restart（重启）。

2. init.zygote32_64.rc

表示既支持 32 位程序也支持 64 位程序，init.zygote32_64.rc 文件的内容如下所示：

```

service zygote /system/bin/app_process32 -Xzygote /system/bin --zygote --start-
system-server --socket-name=zygote
    class main
    priority -20
    user root
    group root readproc
    socket zygote stream 660 root system
    onrestart write /sys/android_power/request_state wake
    ...

```

```

service zygote_secondary /system/bin/app_process64 -Xzygote /system/bin --zygote
--socket-name=zygote_secondary
    class main
    priority -20
    user root
    group root readproc
    socket zygote_secondary stream 660 root system
    onrestart restart zygote
    writepid /dev/cpuset/foreground/tasks

```

脚本中有两个 Service 类型语句，说明会启动两个 Zygote 进程，一个名称为 zygote，执行程序为 app_process32，作为主模式；另一个名称为 zygote_secondary，执行程序为 app_process64，作为辅模式。

剩余的 init.zygote64.rc 和 init.zygote64_32.rc 与上面讲到的 Zygote 启动脚本类似，这里就不再赘述了，我们在 2.1.3 节分析的 Zygote 启动脚本就是支持 64 位程序的 init.zygote64.rc。

2.2.3 Zygote 进程启动过程介绍

从 2.1.5 节中可知 init 启动 Zygote 时主要是调用 app_main.cpp 的 main 函数中的 AppRuntime 的 start 方法来启动 Zygote 进程的，我们就先从 app_main.cpp 的 main 函数开始分析，Zygote 进程启动过程的时序图如图 2-1 所示。

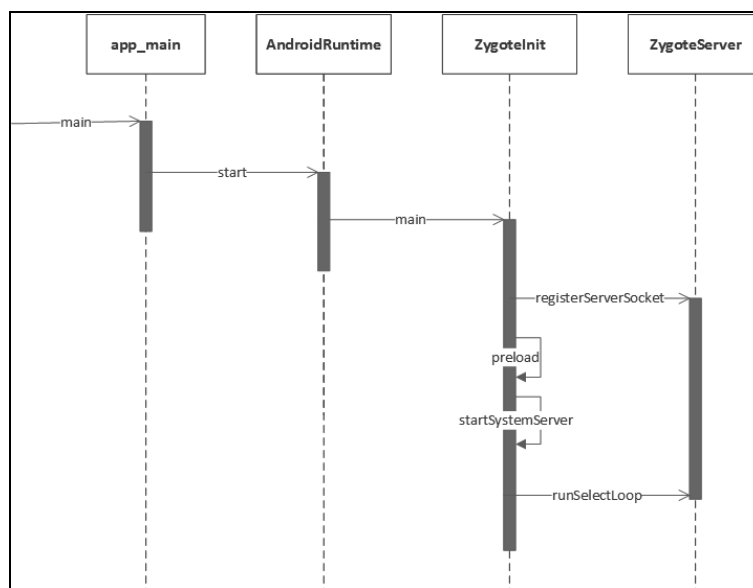


图 2-1 Zygote 进程启动过程的时序图

```
frameworks/base/cmds/app_process/app_main.cpp
```

```
int main(int argc, char* const argv[])
{
    ...
    while (i < argc) {
        const char* arg = argv[i++];
        if (strcmp(arg, "--zygote") == 0) { //1
            //如果当前运行在 Zygote 进程中，则将 zygote 设置为 true
            zygote = true; //2
            niceName = ZYGOTE_NICE_NAME;
        } else if (strcmp(arg, "--start-system-server") == 0) { //3
            //如果当前运行在 SystemServer 进程中，则将 startSystemServer 设置为 true
            startSystemServer = true; //4
        }
        ...
    }
    ...
    if (!niceName.isEmpty()) {
        runtime.setArgv0(niceName.string(), true /* setProcName */);
    }
    //如果运行在 Zygote 进程中
    if (zygote) { //5
        runtime.start("com.android.internal.os.ZygoteInit", args, zygote); //6
    } else if (className) {
        runtime.start("com.android.internal.os.RuntimeInit", args, zygote);
    } else {
        fprintf(stderr, "Error: no class name or --zygote supplied.\n");
        app_usage();
        LOG_ALWAYS_FATAL("app_process: no class name or --zygote supplied.");
    }
}
```

Zygote 进程都是通过 `fork` 自身来创建子进程的，这样 Zygote 进程以及它的子进程都可以进入 `app_main.cpp` 的 `main` 函数，因此 `main` 函数中为了区分当前运行在哪个进程，会在注释 1 处判断参数 `arg` 中是否包含了 “--zygote”，如果包含了则说明 `main` 函数是运行在 Zygote 进程中的并在注释 2 处将 `zygote` 设置为 `true`。同理在注释 3 处判断参数 `arg` 中是否包含了 “--start-system-server”，如果包含了则说明 `main` 函数是运行在 `SystemServer` 进程中的并在注释 4 处将 `startSystemServer` 设置为 `true`。

在注释 5 处，如果 `zygote` 为 `true`，就说明当前运行在 Zygote 进程中，就会调用注释 6 处的 `AppRuntime` 的 `start` 函数，如下所示：

```
frameworks/base/core/jni/AndroidRuntime.cpp
```

```
void AndroidRuntime::start(const char* className, const Vector<String8>& options,
bool zygote)
{
    ...
    JNIEnv* env;
    //启动 Java 虚拟机
    if (startVm(&mJavaVM, &env, zygote) != 0) { //1
        return;
    }
    onVmCreated(env);
    //为 Java 虚拟机注册 JNI 方法
    if (startReg(env) < 0) { //2
        ALOGE("Unable to register all android natives\n");
        return;
    }
    ...
    //从 app_main 的 main 函数得知 className 为 com.android.internal.os.ZygoteInit
    classNameStr = env->NewStringUTF(className); //3
    assert(classNameStr != NULL);
    env->SetObjectArrayElement(strArray, 0, classNameStr);
    for (size_t i = 0; i < options.size(); ++i) {
        jstring optionsStr = env->NewStringUTF(options.itemAt(i).string());
        assert(optionsStr != NULL);
        env->SetObjectArrayElement(strArray, i + 1, optionsStr);
    }
    //将 className 的 "." 替换为 "/"
    char* slashClassName = toSlashClassName(className); //4
    //找到 ZygoteInit
    jclass startClass = env->FindClass(slashClassName); //5
    if (startClass == NULL) {
        ALOGE("JavaVM unable to locate class '%s'\n", slashClassName);
    } else {
        //找到 ZygoteInit 的 main 方法
        jmethodID startMeth = env->GetStaticMethodID(startClass, "main",
            "([Ljava/lang/String;)V"); //6
        if (startMeth == NULL) {
            ALOGE("JavaVM unable to find main() in '%s'\n", className);
            /* keep going */
        } else {
            //通过 JNI 调用 ZygoteInit 的 main 方法
            env->CallStaticVoidMethod(startClass, startMeth, strArray); //7
        }
    }
}
```

```

    #if 0
        if (env->ExceptionCheck())
            threadExitUncaughtException(env);
    #endif
    }
    }
    ...
}

```

在注释 1 处调用 `startVm` 函数来创建 Java 虚拟机,在注释 2 处调用 `startReg` 函数为 Java 虚拟机注册 JNI 方法。注释 3 处的 `className` 的值是传进来的参数,它的值为 `com.android.internal.os.ZygoteInit`。在注释 4 处通过 `toSlashClassName` 函数,将 `className` 的“.”替换为“/”,替换后的值为 `com/android/internal/os/ZygoteInit`,并赋值给 `slashClassName`,接着在注释 5 处根据 `slashClassName` 找到 `ZygoteInit`,找到了 `ZygoteInit` 后顺理成章地在注释 6 处找到 `ZygoteInit` 的 `main` 方法。最终会在注释 7 处通过 JNI 调用 `ZygoteInit` 的 `main` 方法。这里为何要使用 JNI 呢?因为 `ZygoteInit` 的 `main` 方法是由 Java 语言编写的,当前的运行逻辑在 Native 中,这就需要通过 JNI 来调用 Java。这样 `Zygote` 就从 Native 层进入了 Java 框架层。

在我们通过 JNI 调用 `ZygoteInit` 的 `main` 方法后,`Zygote` 便进入了 Java 框架层,此前是没有任何代码进入 Java 框架层的,换句话说就是 `Zygote` 开创了 Java 框架层。该 `main` 方法代码如下:

```
frameworks/base/core/java/com/android/internal/os/ZygoteInit.java
```

```

public static void main(String argv[]) {
    ...
    try {
        ...
        //创建一个 Server 端的 Socket, socketName 的值为“zygote”
        zygoteServer.registerServerSocket(socketName); //1
        if (!enableLazyPreload) {
            bootTimingsTraceLog.traceBegin("ZygotePreload");
            EventLog.writeEvent(LOG_BOOT_PROGRESS_PRELOAD_START,
                SystemClock.uptimeMillis());
            //预加载类和资源
            preload(bootTimingsTraceLog); //2
            EventLog.writeEvent(LOG_BOOT_PROGRESS_PRELOAD_END,
                SystemClock.uptimeMillis());
            bootTimingsTraceLog.traceEnd();
        } else {
            Zygote.resetNicePriority();

```

```

    }
    ...
    if (startSystemServer) {
        //启动 SystemServer 进程
        startSystemServer(abiList, socketName, zygoteServer);//3
    }
    Log.i(TAG, "Accepting command socket connections");
    //等待 AMS 请求
    zygoteServer.runSelectLoop(abiList);//4
    zygoteServer.closeServerSocket();
} catch (Zygote.MethodAndArgsCaller caller) {
    caller.run();
} catch (Throwable ex) {
    Log.e(TAG, "System zygote died with exception", ex);
    zygoteServer.closeServerSocket();
    throw ex;
}
}

```

在注释 1 处通过 registerServerSocket 方法来创建一个 Server 端的 Socket，这个 name 为“zygote”的 Socket 用于等待 ActivityManagerService 请求 Zygote 来创建新的应用程序进程，关于 AMS 将在第 6 章进行介绍。在注释 2 处预加载类和资源。在注释 3 处启动 SystemServer 进程，这样系统的服务也会由 SystemServer 进程启动起来。在注释 4 处调用 ZygoteServer 的 runSelectLoop 方法来等待 AMS 请求创建新的应用程序进程。由此得知，ZygoteInit 的 main 方法主要做了 4 件事：

- (1) 创建一个 Server 端的 Socket。
- (2) 预加载类和资源。
- (3) 启动 SystemServer 进程。
- (4) 等待 AMS 请求创建新的应用程序进程。

其中第二件事预加载类和资源这里不做介绍，有兴趣的读者可以查看源码，这里会对其他的主要事件进行分析。

1. registerZygoteSocket

首先我们来查看 ZygoteServer 的 registerZygoteSocket 方法做了什么，如下所示：

```

frameworks/base/core/java/com/android/internal/os/ZygoteServer.java
void registerServerSocket(String socketName) {

```



```

if (mServerSocket == null) {
    int fileDesc;
    //拼接 Socket 的名称
    final String fullSocketName = ANDROID_SOCKET_PREFIX + socketName;//1
    try {
        //得到 Socket 的环境变量的值
        String env = System.getenv(fullSocketName);//2
        //将 Socket 环境变量的值转换为文件描述符的参数
        fileDesc = Integer.parseInt(env);//3
    } catch (RuntimeException ex) {
        throw new RuntimeException(fullSocketName + " unset or invalid", ex);
    }
    try {
        //创建文件描述符
        FileDescriptor fd = new FileDescriptor();//4
        fd.setInt$(fileDesc);//5
        //创建服务器端 Socket
        mServerSocket = new LocalServerSocket(fd);//6
    } catch (IOException ex) {
        throw new RuntimeException(
            "Error binding to local socket '" + fileDesc + "'", ex);
    }
}
}

```

在注释1处拼接Socket的名称,其中 ANDROID_SOCKET_PREFIX 的值为“ANDROID_SOCKET_”, socketName 的值是传进来的值,等于“zygote”,因此 fullSocketName 的值为“ANDROID_SOCKET_zygote”。在注释2处将 fullSocketName 转换为环境变量的值,再在注释3处转换为文件描述符的参数。在注释4处创建文件描述符,并在注释5处传入此前转换的文件操作符参数。在注释6处创建 LocalServerSocket,也就是服务器端的 Socket,并将文件操作符作为参数传进去。在 Zygote 进程将 SystemServer 进程启动后,就会在这个服务器端的 Socket 上等待 AMS 请求 Zygote 进程来创建新的应用程序进程。

2. 启动 SystemServer 进程

接下来查看 startSystemServer 函数,代码如下所示:

```
frameworks/base/core/java/com/android/internal/os/ZygoteInit.java
```

```

private static boolean startSystemServer(String abiList, String socketName,
    ZygoteServer zygoteServer)
    throws Zygote.MethodAndArgsCaller, RuntimeException {
    ...
}

```

```

//创建 args 数组, 这个数组用来保存启动 SystemServer 的启动参数
/*1*/
String args[] = {
    "--setuid=1000",
    "--setgid=1000",
    "--setgroups=1001,1002,1003,1004,1005,1006,1007,1008,1009,1010,1018,
    1021,1023,1032,3001,3002,3003,3006,3007,3009,3010",
    "--capabilities=" + capabilities + "," + capabilities,
    "--nice-name=system_server",
    "--runtime-args",
    "com.android.server.SystemServer",
};
ZygoteConnection.Arguments parsedArgs = null;
int pid;
try {
    parsedArgs = new ZygoteConnection.Arguments(args);//2
    ZygoteConnection.applyDebuggerSystemProperty(parsedArgs);
    ZygoteConnection.applyInvokeWithSystemProperty(parsedArgs);
    /**
     * 3 创建一个子进程, 也就是 SystemServer 进程
     */
    pid = Zygote.forkSystemServer(
        parsedArgs.uid, parsedArgs.gid,
        parsedArgs.gids,
        parsedArgs.debugFlags,
        null,
        parsedArgs.permittedCapabilities,
        parsedArgs.effectiveCapabilities);
} catch (IllegalArgumentException ex) {
    throw new RuntimeException(ex);
}
//当前代码逻辑运行在子进程中
if (pid == 0) {
    if (hasSecondZygote(abiList)) {
        waitForSecondaryZygote(socketName);
    }
    zygoteServer.closeServerSocket();
    //处理 SystemServer 进程
    handleSystemServerProcess(parsedArgs);//4
}
return true;
}

```

注释 1 处的代码用来创建 args 数组, 这个数组用来保存启动 SystemServer 的启动参数,

其中可以看出 SystemServer 进程的用户 id 和用户组 id 被设置为 1000，并且拥有用户组 1001~1010、1018、1021、1032、3001~3010 的权限；进程名为 system_server；启动的类名为 com.android.server.SystemServer。在注释 2 处将 args 数组封装成 Arguments 对象并供注释 3 处的 forkSystemServer 函数调用。在注释 3 处调用 Zygote 的 forkSystemServer 方法，其内部会调用 nativeForkSystemServer 这个 Native 方法，nativeForkSystemServer 方法最终会通过 fork 函数在当前进程创建一个子进程，也就是 SystemServer 进程，如果 forkSystemServer 方法返回的 pid 的值为 0，就表示当前的代码运行在新创建的子进程中，则执行注释 4 处的 handleSystemServerProcess 来处理 SystemServer 进程，关于 SystemServer 进程启动会在 2.3 节进行介绍。

3. runSelectLoop

启动 SystemServer 进程后，会执行 ZygoteServer 的 runSelectLoop 方法，如下所示：

frameworks/base/core/java/com/android/internal/os/ZygoteServer.java

```
void runSelectLoop(String abiList) throws Zygote.MethodAndArgsCaller {
    ArrayList<FileDescriptor> fds = new ArrayList<FileDescriptor>();
    ArrayList<ZygoteConnection> peers = new ArrayList<ZygoteConnection>();
    fds.add(mServerSocket.getFileDescriptor());//1
    peers.add(null);
    //无限循环等待 AMS 的请求
    while (true) {
        StructPollfd[] pollFds = new StructPollfd[fds.size()];
        for (int i = 0; i < pollFds.length; ++i) {//2
            pollFds[i] = new StructPollfd();
            pollFds[i].fd = fds.get(i);
            pollFds[i].events = (short) POLLIN;
        }
        try {
            Os.poll(pollFds, -1);
        } catch (ErrnoException ex) {
            throw new RuntimeException("poll failed", ex);
        }
        for (int i = pollFds.length - 1; i >= 0; --i) {//3
            if ((pollFds[i].revents & POLLIN) == 0) {
                continue;
            }
            if (i == 0) {
                ZygoteConnection newPeer = acceptCommandPeer(abiList);//4
                peers.add(newPeer);
                fds.add(newPeer.getFileDescriptor());
            }
        }
    }
}
```

```

        } else {
            boolean done = peers.get(i).runOnce(this); //5
            if (done) {
                peers.remove(i);
                fds.remove(i);
            }
        }
    }
}
}

```

注释 1 处的 `mServerSocket` 就是我们在 `registerZygoteSocket` 函数中创建的服务器端 `Socket`, 调用 `mServerSocket.getFileDescriptor()` 函数用来获得该 `Socket` 的 `fd` 字段的值并添加到 `fd` 列表 `fds` 中。接下来无限循环用来等待 AMS 请求 `Zygote` 进程创建新的应用程序进程。在注释 2 处通过遍历将 `fds` 存储的信息转移到 `pollFds` 数组中。在注释 3 处对 `pollFds` 进行遍历, 如果 `i==0`, 说明服务器端 `Socket` 与客户端连接上了, 换句话说就是, 当前 `Zygote` 进程与 AMS 建立了连接。在注释 4 处通过 `acceptCommandPeer` 方法得到 `ZygoteConnection` 类并添加到 `Socket` 连接列表 `peers` 中, 接着将该 `ZygoteConnection` 的 `fd` 添加到 `fd` 列表 `fds` 中, 以便可以接收到 AMS 发送过来的请求。如果 `i` 的值不等于 0, 则说明 AMS 向 `Zygote` 进程发送了一个创建应用进程的请求, 则在注释 5 处调用 `ZygoteConnection` 的 `runOnce` 函数来创建一个新的应用程序进程, 并在成功创建后将这个连接从 `Socket` 连接列表 `peers` 和 `fd` 列表 `fds` 中清除。

2.2.4 Zygote进程启动总结

`Zygote` 进程启动共做了如下几件事:

- (1) 创建 `AppRuntime` 并调用其 `start` 方法, 启动 `Zygote` 进程。
- (2) 创建 Java 虚拟机并为 Java 虚拟机注册 JNI 方法。
- (3) 通过 JNI 调用 `ZygoteInit` 的 `main` 函数进入 `Zygote` 的 Java 框架层。
- (4) 通过 `registerZygoteSocket` 方法创建服务器端 `Socket`, 并通过 `runSelectLoop` 方法等待 AMS 的请求来创建新的应用程序进程。
- (5) 启动 `SystemService` 进程。

2.3 SystemServer处理过程

SystemServer 进程主要用于创建系统服务，我们熟知的 AMS、WMS 和 PMS 都是由它来创建的，本书后面的章节会介绍 AMS 和 WMS，因此掌握 SystemServer 进程是如何启动的，它在启动时做了哪些工作是十分必要的。

2.3.1 Zygote处理SystemServer进程

在 2.2 节中讲到了 Zygote 进程启动了 SystemServer 进程，本节来学习 Zygote 是如何处理 SystemServer 进程的，时序图如图 2-2 所示。

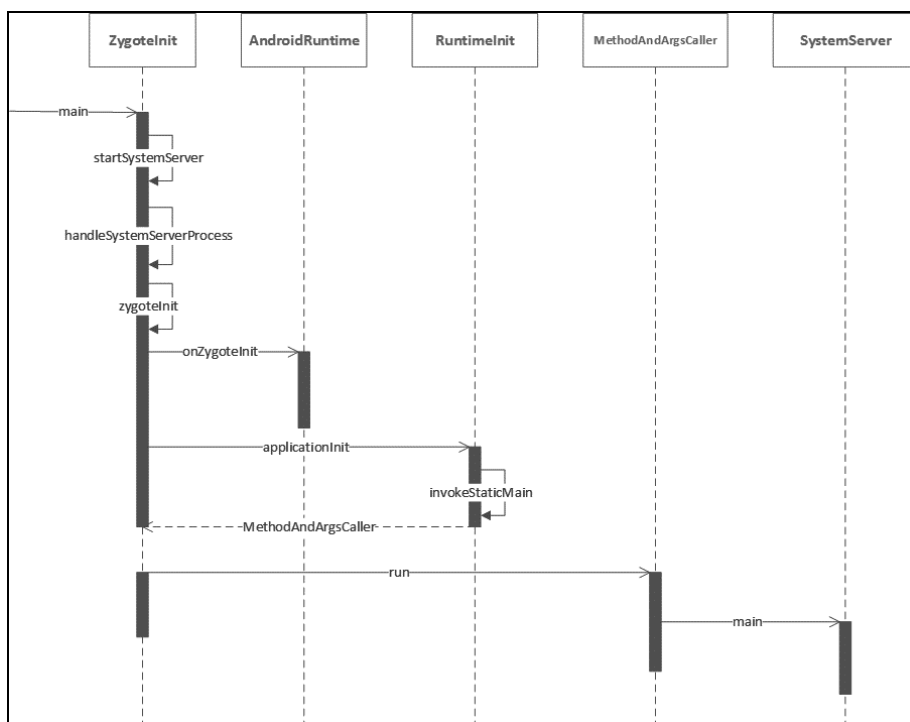


图 2-2 Zygote 处理 SystemServer 进程的时序图

在 ZygoteInit.java 的 startSystemServer 方法中启动了 SystemServer 进程，如下所示：

```
frameworks/base/core/java/com/android/internal/os/ZygoteInit.java
```

```
private static boolean startSystemServer(String abiList, String socketName)
    throws MethodAndArgsCaller, RuntimeException {
    ...
}
```

```

//当前运行在 SystemServer 进程中
if (pid == 0) {
    if (hasSecondZygote(abiList)) {
        waitForSecondaryZygote(socketName);
    }
    //关闭 Zygote 进程创建的 Socket
    zygoteServer.closeServerSocket();//1
    handleSystemServerProcess(parsedArgs);//2
}
return true;
}

```

SystemServer 进程复制了 Zygote 进程的地址空间，因此也会得到 Zygote 进程创建的 Socket，这个 Socket 对于 SystemServer 进程没有用处，因此，需要注释 1 处的代码来关闭该 Socket，接着在注释 2 处调用 handleSystemServerProcess 方法来启动 SystemServer 进程。handleSystemServerProcess 方法的代码如下所示：

frameworks/base/core/java/com/android/internal/os/ZygoteInit.java

```

private static void handleSystemServerProcess(
    ZygoteConnection.Arguments parsedArgs)
    throws Zygote.MethodAndArgsCaller {
    ...
    if (parsedArgs.invokeWith != null) {
        ...
    } else {
        ClassLoader cl = null;
        if (systemServerClasspath != null) {
            cl = createPathClassLoader(systemServerClasspath, parsedArgs.
targetSdkVersion);//1
            Thread.currentThread().setContextClassLoader(cl);
        }
        ZygoteInit.zygoteInit(parsedArgs.targetSdkVersion, parsedArgs.remainingArgs,
cl);//2
    }
}

```

在注释 1 处创建了 PathClassLoader，关于 PathClassLoader 将在第 12 章进行介绍。在注释 2 处调用了 ZygoteInit 的 zygoteInit 方法，代码如下所示：

frameworks/base/core/java/com/android/internal/os/ZygoteInit.java

```

public static final void zygoteInit(int targetSdkVersion, String[] argv,
    ClassLoader classLoader) throws Zygote.MethodAndArgsCaller {
    if (RuntimeInit.DEBUG) {

```

```

        Slog.d(RuntimeInit.TAG, "RuntimeInit: Starting application from zygote");
    }
    Trace.traceBegin(Trace.TRACE_TAG_ACTIVITY_MANAGER, "ZygoteInit");
    RuntimeInit.redirectLogStreams();
    RuntimeInit.commonInit();
    //启动 Binder 线程池
    ZygoteInit.nativeZygoteInit();//1
    //进入 SystemServer 的 main 方法
    RuntimeInit.applicationInit(targetSdkVersion, argv, classLoader);//2
}

```

在注释 1 处调用 nativeZygoteInit 方法，一看方法的名称就知道调用的是 Native 层的代码，用来启动 Binder 线程池，这样 SystemServer 进程就可以使用 Binder 与其他进程进行通信了。注释 2 处是用于进入 SystemServer 的 main 方法，现在分别对注释 1 和注释 2 的内容进行介绍。

1. 启动 Binder 线程池

nativeZygoteInit 是一个 Native 方法，因此我们先要了解它对应的 JNI 文件，不了解 JNI 的可以查看第 9 章内容，如下所示：

frameworks/base/core/jni/AndroidRuntime.cpp

```

int register_com_android_internal_os_ZygoteInit(JNIEnv* env)
{
    const JNINativeMethod methods[] = {
        { "nativeZygoteInit", "()V",
          (void*) com_android_internal_os_ZygoteInit_nativeZygoteInit },
    };
    return jniRegisterNativeMethods(env, "com/android/internal/os/ZygoteInit",
        methods, NELEM(methods));
}

```

通过 JNI 的 gMethods 数组，可以看出 nativeZygoteInit 方法对应的是 JNI 文件 AndroidRuntime.cpp 的 com_android_internal_os_ZygoteInit_nativeZygoteInit 函数：

frameworks/base/core/jni/AndroidRuntime.cpp

```

static void com_android_internal_os_ZygoteInit_nativeZygoteInit(JNIEnv* env,
    jobject clazz)
{
    gCurRuntime->onZygoteInit();
}

```

这里 gCurRuntime 是 AndroidRuntime 类型的指针，具体指向的是 AndroidRuntime 的子

类 AppRuntime，它在 app_main.cpp 中定义，我们接着来查看 AppRuntime 的 onZygoteInit 方法，代码如下所示：

```
frameworks/base/cmds/app_process/app_main.cpp
```

```
virtual void onZygoteInit()
{
    sp<ProcessState> proc = ProcessState::self();
    ALOGV("App process: starting thread pool.\n");
    proc->startThreadPool();//1
}
```

注释 1 处的代码用来启动一个 Binder 线程池，这样 SystemServer 进程就可以使用 Binder 与其他进程进行通信了。看到这里我们知道 RuntimeInit.java 的 nativeZygoteInit 函数主要是用来启动 Binder 线程池的。

2. 进入 SystemServer 的 main 方法

我们再回到 RuntimeInit.java 的代码，在注释 2 处调用了 RuntimeInit 的 applicationInit 方法，代码如下所示：

```
frameworks/base/core/java/com/android/internal/os/RuntimeInit.java
```

```
protected static void applicationInit(int targetSdkVersion, String[] argv,
ClassLoader classLoader)
    throws Zygote.MethodAndArgsCaller {
    ...
    invokeStaticMain(args.startClass, args.startArgs, classLoader);
}
```

在 applicationInit 方法中主要调用了 invokeStaticMain 方法：

```
frameworks/base/core/java/com/android/internal/os/RuntimeInit.java
```

```
private static void invokeStaticMain(String className, String[] argv, ClassLoader
classLoader)
    throws Zygote.MethodAndArgsCaller {
    Class<?> cl;

    try {
        //通过反射得到 SystemServer 类
        cl = Class.forName(className, true, classLoader);//1
    } catch (ClassNotFoundException ex) {
        throw new RuntimeException(
            "Missing class when invoking static main " + className,
            ex);
    }
```



```

    }
    Method m;
    try {
        //找到 SystemServer 的 main 方法
        m = cl.getMethod("main", new Class[] { String[].class });//2
    } catch (NoSuchMethodException ex) {
        throw new RuntimeException(
            "Missing static main on " + className, ex);
    } catch (SecurityException ex) {
        throw new RuntimeException(
            "Problem getting static main on " + className, ex);
    }
    int modifiers = m.getModifiers();
    if (!(Modifier.isStatic(modifiers) && Modifier.isPublic(modifiers))) {
        throw new RuntimeException(
            "Main method is not public and static on " + className);
    }
    throw new Zygote.MethodAndArgsCaller(m, argv);//3
}

```

注释 1 处的 `className` 为 `com.android.server.SystemServer`，通过反射返回的 `cl` 为 `SystemServer` 类。在注释 2 处找到 `SystemServer` 中的 `main` 方法。在注释 3 处将找到的 `main` 方法传入 `MethodAndArgsCaller` 异常中并抛出该异常，捕获 `MethodAndArgsCaller` 异常的代码在 `ZygoteInit.java` 的 `main` 方法中，这个 `main` 方法会调用 `SystemServer` 的 `main` 方法。那么为什么不直接在 `invokeStaticMain` 方法中调用 `SystemServer` 的 `main` 方法呢？原因是这种抛出异常的处理会清除所有的设置过程需要的堆栈帧，并让 `SystemServer` 的 `main` 方法看起来像是 `SystemServer` 进程的入口方法。在 `Zygote` 启动了 `SystemServer` 进程后，`SystemServer` 进程已经做了很多的准备工作，而这些工作都是在 `SystemServer` 的 `main` 方法调用之前做的，这使得 `SystemServer` 的 `main` 方法看起来不像是 `SystemServer` 进程的入口方法，而这种抛出异常交由 `ZygoteInit.java` 的 `main` 方法来处理，会让 `SystemServer` 的 `main` 方法看起来像是 `SystemServer` 进程的入口方法。

下面来查看在 `ZygoteInit.java` 的 `main` 方法中是如何捕获 `MethodAndArgsCaller` 异常的：

```
frameworks/base/core/java/com/android/internal/os/ZygoteInit.java
```

```

public static void main(String argv[]) {
    ...
    closeServerSocket();
} catch (MethodAndArgsCaller caller) {
    caller.run();//1
} catch (RuntimeException ex) {

```

```
        Log.e(TAG, "Zygote died with exception", ex);
        closeServerSocket();
        throw ex;
    }
}
```

当捕获到 MethodAndArgsCaller 异常时就会在注释 1 处调用 MethodAndArgsCaller 的 run 方法，MethodAndArgsCaller 是 Zygote.java 的静态内部类：

frameworks/base/core/java/com/android/internal/os/Zygote.java

```
public static class MethodAndArgsCaller extends Exception
    implements Runnable {
    private final Method mMethod;
    private final String[] mArgs;
    public MethodAndArgsCaller(Method method, String[] args) {
        mMethod = method;
        mArgs = args;
    }
    public void run() {
        try {
            mMethod.invoke(null, new Object[] { mArgs });//1
        } catch (IllegalAccessException ex) {
            throw new RuntimeException(ex);
        }
        ...
    }
}
```

注释 1 处的 mMethod 指的就是 SystemServer 的 main 方法，调用了 mMethod 的 invoke 方法后，SystemServer 的 main 方法就会被动态调用，SystemServer 进程就进入了 SystemServer 的 main 方法中。

2.3.2 解析SystemServer进程

下面来查看 SystemServer 的 main 方法：

frameworks/base/services/java/com/android/server/SystemServer.java

```
public static void main(String[] args) {
    new SystemServer().run();
}
```

main 方法中只调用了 SystemServer 的 run 方法，如下所示：

```
frameworks/base/services/java/com/android/server/SystemServer.java
```

```
private void run() {
    try {
        ...
        //创建消息 Looper
        Looper.prepareMainLooper();
        //加载了动态库 libandroid_servers.so
        System.loadLibrary("android_servers");//1
        performPendingShutdown();
        // 创建系统的 Context
        createSystemContext();
        mSystemServiceManager = new SystemServiceManager(mSystemContext);//2
        mSystemServiceManager.setRuntimeRestarted(mRuntimeRestart);
        LocalServices.addService(SystemServiceManager.class, mSystemServiceManager);
        SystemServerInitThreadPool.get();
    } finally {
        traceEnd();
    }
    try {
        traceBeginAndSlog("StartServices");
        //启动引导服务
        startBootstrapServices();//3
        //启动核心服务
        startCoreServices();//4
        //启动其他服务
        startOtherServices();//5
        SystemServerInitThreadPool.shutdown();
    } catch (Throwable ex) {
        Slog.e("System", "*****");
        Slog.e("System", "***** Failure starting system services", ex);
        throw ex;
    } finally {
        traceEnd();
    }
    ...
}
```

在注释 1 处加载了动态库 libandroid_servers.so。接下来在注释 2 处创建 SystemServiceManager，它会对系统服务进行创建、启动和生命周期管理。在注释 3 处的 startBootstrapServices 方法中用 SystemServiceManager 启动了 ActivityManagerService、PowerManagerService、PackageManagerService 等服务。在注释 4 处的 startCoreServices 方法中则启动了 DropBoxManagerService、BatteryService、UsageStatsService 和 WebViewUpdateService。

在注释 5 处的 `startOtherServices` 方法中启动了 `CameraService`、`AlarmManagerService`、`VrManagerService` 等服务。这些服务的父类均为 `SystemService`。从注释 3、4、5 的方法中可以看出，官方把系统服务分为了三种类型，分别是引导服务、核心服务和其他服务，其中其他服务是一些非紧要和不需立即启动的服务。这些系统服务总共有 100 多个，表 2-1 列出部分系统服务及其作用。

表 2-1 部分系统服务及其作用

引导服务	作用
Installer	系统安装 APK 时的一个服务类，启动完成 Installer 服务之后才能启动其他的系统服务
ActivityManagerService	负责四大组件的启动、切换、调度
PowerManagerService	计算系统中和 Power 相关的计算，然后决策系统应该如何反应
LightsService	管理和显示背光 LED
DisplayManagerService	用来管理所有显示设备
UserManagerService	多用户模式管理
SensorService	为系统提供各种感应器服务
PackageManagerService	用来对 APK 进行安装、解析、删除、卸载等操作
.....
核心服务	
DropBoxManagerService	用于生成和管理系统运行时的一些日志文件
BatteryService	管理电池相关的服务
UsageStatsService	收集用户使用每一个 App 的频率、使用时长
WebViewUpdateService	WebView 更新服务
其他服务	
CameraService	摄像头相关服务
AlarmManagerService	全局定时器管理服务
InputManagerService	管理输入事件
WindowManagerService	窗口管理服务
VrManagerService	VR 模式管理服务
BluetoothService	蓝牙管理服务
NotificationManagerService	通知管理服务
DeviceStorageMonitorService	存储相关管理服务
LocationManagerService	定位管理服务
AudioService	音频相关管理服务
.....

这些系统服务启动逻辑是相似的，这里以启动 `PowerManagerService` 来进行举例，代码如下所示：

```
mPowerManagerService=mSystemServiceManager.startService(PowerManagerService.class);
```

`SystemServiceManager` 的 `startService` 方法启动了 `PowerManagerService`，`startService` 方法如下所示：

```
frameworks/base/services/core/java/com/android/server/SystemServiceManager.java
```

```
public void startService(@NonNull final SystemService service) {
    //注册 Service
    mServices.add(service); //1
    long time = System.currentTimeMillis();
    try {
        //启动 Service
        service.onStart(); //2
    } catch (RuntimeException ex) {
        throw new RuntimeException("Failed to start service " + service.getClass().
            getName() + ": onStart threw an exception", ex);
    }
    warnIfTooLong(System.currentTimeMillis() - time, service, "onStart");
}
```

在注释 1 处将 `PowerManagerService` 添加到 `mServices` 中，其中 `mServices` 是一个存储 `SystemService` 类型的 `ArrayList`，这样就完成了 `PowerManagerService` 的注册工作。在注释 2 处调用 `PowerManagerService` 的 `onStart` 函数完成启动 `PowerManagerService`。

除了用 `mSystemServiceManager` 的 `startService` 函数来启动系统服务外，也可以通过如下形式来启动系统服务，以 `PackageManagerService` 为例：

```
mPackageManagerService = PackageManagerService.main(mSystemContext, installer,
mFactoryTestMode != FactoryTest.FACTORY_TEST_OFF, mOnlyCore);
```

直接调用了 `PackageManagerService` 的 `main` 方法：

```
frameworks/base/services/core/java/com/android/server/pm/PackageManagerService.java
```

```
public static PackageManagerService main(Context context, Installer installer,
    boolean factoryTest, boolean onlyCore) {
    //自检初始的设置
    PackageManagerServiceCompilerMapping.checkProperties();
    PackageManagerService m = new PackageManagerService(context, installer,
        factoryTest, onlyCore); //1
    m.enableSystemUserPackages();
```

```
        ServiceManager.addService("package", m); //2
    return m;
}
```

在注释 1 处直接创建 `PackageManagerService` 并在注释 2 处将 `PackageManagerService` 注册到 `ServiceManager` 中，`ServiceManager` 用来管理系统中的各种 `Service`，用于系统 C/S 架构中的 Binder 通信机制：Client 端要使用某个 `Service`，则需要先到 `ServiceManager` 查询 `Service` 的相关信息，然后根据 `Service` 的相关信息与 `Service` 所在的 Server 进程建立通信通路，这样 Client 端就可以使用 `Service` 了。

2.3.3 SystemServer进程总结

`SystemServer` 进程被创建后，主要做了如下工作：

- (1) 启动 Binder 线程池，这样就可以与其他进程进行通信。
- (2) 创建 `SystemServiceManager`，其用于对系统的服务进行创建、启动和生命周期管理。
- (3) 启动各种系统服务。

2.4 Launcher启动过程

此前已经学习了 Android 系统启动流程的 `init` 进程、`Zygote` 进程和 `SystemServer` 进程，最后我们来学习 `Launcher` 的启动过程。

2.4.1 Launcher概述

系统启动的最后一步是启动一个应用程序用来显示系统中已经安装的应用程序，这个应用程序就叫作 `Launcher`。`Launcher` 在启动过程中会请求 `PackageManagerService` 返回系统中已经安装的应用程序的信息，并将这些信息封装成一个快捷图标列表显示在系统屏幕上，这样用户可以通过点击这些快捷图标来启动相应的应用程序。

通俗来讲 `Launcher` 就是 Android 系统的桌面，它的作用主要有以下两点：

- (1) 作为 Android 系统的启动器，用于启动应用程序。
- (2) 作为 Android 系统的桌面，用于显示和管理应用程序的快捷图标或者其他桌面组件。

2.4.2 Launcher启动过程介绍

SystemService 进程在启动的过程中会启动 PackageManagerService, PackageManagerService 启动后会将系统中的应用程序安装完成。在此前已经启动的 AMS 会将 Launcher 启动起来。Launcher 启动过程的时序图如图 2-3 所示。

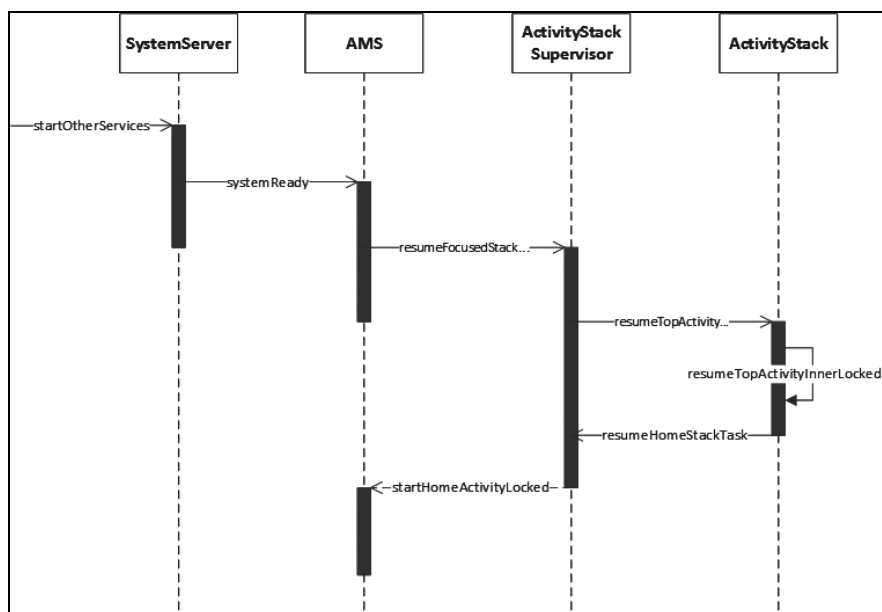


图 2-3 Launcher 启动过程时序图

启动 Launcher 的入口为 AMS 的 systemReady 方法，它在 SystemServer 的 startOtherServices 方法中被调用，如下所示：

```
frameworks/base/services/java/com/android/server/SystemServer.java
```

```

private void startOtherServices() {
    ...
    mActivityManagerService.systemReady() -> { //1
        Slog.i(TAG, "Making services ready");
        traceBeginAndSlog("StartActivityManagerReadyPhase");
        mSystemServiceManager.startBootPhase(
            SystemService.PHASE_ACTIVITY_MANAGER_READY);
    }
    ...
}

```

与 Android 7.1.2 源码不同的是，Android 8.0 的部分源码引入了 Java Lambda 表达式，比如注释 1 处。下面来查看 AMS 的 `systemReady` 方法做了什么：

```
frameworks/base/services/core/java/com/android/server/am/ActivityManagerService.java
```

```
public void systemReady(final Runnable goingCallback) {
    ...
    synchronized (this) {
        ...
        mStackSupervisor.resumeFocusedStackTopActivityLocked();
        mUserController.sendUserSwitchBroadcastsLocked(-1, currentUserId);
    }
}
```

`systemReady` 方法中调用了 `ActivityStackSupervisor` 的 `resumeFocusedStackTopActivityLocked` 方法：

```
frameworks/base/services/core/java/com/android/server/am/ActivityStackSupervisor.java
```

```
boolean resumeFocusedStackTopActivityLocked(
    ActivityStack targetStack, ActivityRecord target, ActivityOptions
    targetOptions) {
    if (targetStack != null && isFocusedStack(targetStack)) {
        return targetStack.resumeTopActivityUncheckedLocked(target,
            targetOptions); //1
    }
    ...
    return false;
}
```

在注释 1 处调用 `ActivityStack` 的 `resumeTopActivityUncheckedLocked` 方法，`ActivityStack` 对象是用来描述 Activity 堆栈的，`resumeTopActivityUncheckedLocked` 方法如下所示：

```
frameworks/base/services/core/java/com/android/server/am/ActivityStack.java
```

```
boolean resumeTopActivityUncheckedLocked(ActivityRecord prev, ActivityOptions
options) {
    if (mStackSupervisor.inResumeTopActivity) {
        return false;
    }
    boolean result = false;
    try {
        mStackSupervisor.inResumeTopActivity = true;
        result = resumeTopActivityInnerLocked(prev, options); //1
    } finally {
        mStackSupervisor.inResumeTopActivity = false;
    }
}
```



```

    }
    mStackSupervisor.checkReadyForSleepLocked();
    return result;
}

```

在注释 1 处调用了 `resumeTopActivityInnerLocked` 方法，如下所示：

frameworks/base/services/core/java/com/android/server/am/ActivityStack.java.java

```

private boolean resumeTopActivityInnerLocked(ActivityRecord prev, ActivityOptions
options) {
    ...
    return isOnHomeDisplay() &&
        mStackSupervisor.resumeHomeStackTask(returnTaskType, prev, "prevFinished");
    ...
}

```

`resumeTopActivityInnerLocked` 方法的代码很长，在此仅截取我们要分析的关键的部分，调用 `ActivityStackSupervisor` 的 `resumeHomeStackTask` 方法，代码如下所示：

frameworks/base/services/core/java/com/android/server/am/ActivityStackSupervisor.java

```

boolean resumeHomeStackTask(ActivityRecord prev, String reason) {
    ...
    if (r != null && !r.finishing) {
        moveFocusableActivityStackToFrontLocked(r, myReason);
        return resumeFocusedStackTopActivityLocked(mHomeStack, prev, null);
    }
    return mService.startHomeActivityLocked(mCurrentUser, myReason);
}

```

在 `resumeHomeStackTask` 方法中调用了 AMS 的 `startHomeActivityLocked` 方法，如下所示：

frameworks/base/services/core/java/com/android/server/am/ActivityManagerService.java

```

boolean startHomeActivityLocked(int userId, String reason) {
    //判断工厂模式和 mTopAction 的值，符合要求就继续执行下去
    if (mFactoryTest == FactoryTest.FACTORY_TEST_LOW_LEVEL
        && mTopAction == null) { //1
        return false;
    }
    //创建 Launcher 启动所需的 Intent
    Intent intent = getHomeIntent();//2
    ActivityInfo aInfo = resolveActivityInfo(intent, STOCK_PM_FLAGS, userId);
    if (aInfo != null) {
        intent.setComponent(new ComponentName(aInfo.applicationInfo.packageName,
            aInfo.name));
    }
}

```

```

        aInfo = new ActivityInfo(aInfo);
        aInfo.applicationInfo = getAppInfoForUser(aInfo.applicationInfo, userId);
        ProcessRecord app = getProcessRecordLocked(aInfo.processName,
            aInfo.applicationInfo.uid, true);
        if (app == null || app.instr == null) { //3
            intent.setFlags(intent.getFlags() | Intent.FLAG_ACTIVITY_NEW_TASK);
            final int resolvedUserId = UserHandle.getUserId(aInfo.applicationInfo.uid);
            final String myReason = reason + ":" + userId + ":" + resolvedUserId;
            //启动 Launcher
            mActivityStarter.startHomeActivityLocked(intent, aInfo, myReason); //4
        }
    } else {
        Slog.wtf(TAG, "No home screen found for " + intent, new Throwable());
    }
    return true;
}

```

注释 1 处的 `mFactoryTest` 代表系统的运行模式，系统的运行模式分为三种，分别是非工厂模式、低级工厂模式和高级工厂模式，`mTopAction` 则用来描述第一个被启动 Activity 组件的 Action，它的默认值为 `Intent.ACTION_MAIN`。因此注释 1 处的代码的意思就是 `mFactoryTest` 为 `FactoryTest.FACTORY_TEST_LOW_LEVEL`（低级工厂模式）并且 `mTopAction` 等于 `null` 时，直接返回 `false`。注释 2 处的 `getHomeIntent` 方法如下所示：

frameworks/base/services/core/java/com/android/server/am/ActivityManagerService.java

```

Intent getHomeIntent() {
    Intent intent = new Intent(mTopAction, mTopData != null ? Uri.parse(mTopData) :
        null);
    intent.setComponent(mTopComponent);
    intent.addFlags(Intent.FLAG_DEBUG_TRIAGED_MISSING);
    if (mFactoryTest != FactoryTest.FACTORY_TEST_LOW_LEVEL) {
        intent.addCategory(Intent.CATEGORY_HOME);
    }
    return intent;
}

```

在 `getHomeIntent` 方法中创建了 `Intent`，并将 `mTopAction` 和 `mTopData` 传入。`mTopAction` 的值为 `Intent.ACTION_MAIN`，并且如果系统运行模式不是低级工厂模式，则将 `intent` 的 `Category` 设置为 `Intent.CATEGORY_HOME`，最后返回该 `Intent`，再回到 AMS 的 `startHomeActivityLocked` 方法，假设系统的运行模式不是低级工厂模式，在注释 3 处判断符合 Action 为 `Intent.ACTION_MAIN`、Category 为 `Intent.CATEGORY_HOME` 的应用程序是否已经启动，如果没启动则调用注释 4 处的方法启动该应用程序。这个被启动的应用程

序就是 Launcher, 因为 Launcher 的 AndroidManifest 文件中的 intent-filter 标签匹配了 Action 为 Intent.ACTION_MAIN, Category 为 Intent.CATEGORY_HOME。

Launcher 的 AndroidManifest 文件如下所示:

packages/apps/Launcher3/AndroidManifest.xml

```
<manifest
  xmlns:android="http://schemas.android.com/apk/res/android"
  package="com.android.launcher3">
  <uses-sdk android:targetSdkVersion="23" android:minSdkVersion="21"/>
  ...
  <application
    ...
    <activity
      android:name="com.android.launcher3.Launcher"
      android:launchMode="singleTask"
      android:clearTaskOnLaunch="true"
      android:stateNotNeeded="true"
      android:windowSoftInputMode="adjustPan|stateUnchanged"
      android:screenOrientation="nosensor"
      android:configChanges="keyboard|keyboardHidden|navigation"
      android:resizeableActivity="true"
      android:resumeWhilePausing="true"
      android:taskAffinity=""
      android:enabled="true">
      <intent-filter>
        <action android:name="android.intent.action.MAIN" />
        <category android:name="android.intent.category.HOME" />
        <category android:name="android.intent.category.DEFAULT" />
        <category android:name="android.intent.category.MONKEY"/>
      </intent-filter>
    </activity>
    ...
  </application>
</manifest>
```

可以看到 intent-filter 设置了 android.intent.category.HOME 属性, 这样名称为 com.android.launcher3.Launcher 的 Activity 就成为了主 Activity。从前面 AMS 的 startHomeActivityLocked 方法的注释 4 处, 我们得知如果 Launcher 没有启动就会调用 ActivityStarter 的 startHomeActivityLocked 方法来启动 Launcher, 如下所示:

frameworks/base/services/core/java/com/android/server/am/ActivityStarter.java

```
void startHomeActivityLocked(Intent intent, ActivityInfo aInfo, String reason) {
```

```

//将 Launcher 放入 HomeStack 中
mSupervisor.moveHomeStackTaskToTop(reason);//1
mLastHomeActivityResult = startActivityLocked(null /*caller*/, intent,
    null /*ephemeralIntent*/, null /*resolvedType*/, aInfo, null /*rInfo*/,
    null /*voiceSession*/, null /*voiceInteractor*/, null /*resultTo*/,
    null /*resultWho*/, 0 /*requestCode*/, 0 /*callingPid*/, 0 /*callingUid*/,
    null /*callingPackage*/, 0 /*realCallingPid*/, 0 /*realCallingUid*/,
    0 /*startFlags*/, null /*options*/, false /*ignoreTargetSecurity*/,
    false /*componentSpecified*/, mLastHomeActivityResultRecord /*outActivity*/,
    null /*container*/, null /*inTask*/, "startHomeActivity: " + reason);
...
}

```

在注释 1 处将 Launcher 放入 HomeStack 中，HomeStack 是在 ActivityStackSupervisor 中定义的用于存储 Launcher 的变量。接着调用 startActivityLocked 方法来启动 Launcher，剩余的过程会和 Activity 的启动过程类似，本书将在第 4 章介绍该知识点。最终进入 Launcher 的 onCreate 方法中，到这里 Launcher 就完成了启动。

2.4.3 Launcher 中应用图标显示过程

Launcher 完成启动后会做很多的工作，作为桌面它会显示应用程序图标，这与应用程序开发有所关联，应用程序图标是用户进入应用程序的入口，因此我们有必要了解 Launcher 是如何显示应用程序图标的。

我们先从 Launcher 的 onCreate 方法入手，代码如下所示：

```
packages/apps/Launcher3/src/com/android/launcher3/Launcher.java
```

```

@Override
protected void onCreate(Bundle savedInstanceState) {
    ...
    LauncherAppState app = LauncherAppState.getInstance();//1
    mDeviceProfile = getResources().getConfiguration().orientation
        == Configuration.ORIENTATION_LANDSCAPE ?
        app.getInvariantDeviceProfile().landscapeProfile
        : app.getInvariantDeviceProfile().portraitProfile;
    mSharedPreferences = Utilities.getPrefs(this);
    mIsSafeModeEnabled = getPackageManager().isSafeMode();
    mModel = app.setLauncher(this);//2
    ....
    if (!mRestoring) {
        if (DISABLE_SYNCHRONOUS_BINDING_CURRENT_PAGE) {
            mModel.startLoader(PagedView.INVALID_RESTORE_PAGE);//3

```

```

        } else {
            mModel.startLoader(mWorkspace.getRestorePage());
        }
    }
    ...
}

```

在注释 1 处获取 LauncherAppState 的实例，在注释 2 处调用它的 setLauncher 方法并将 Launcher 对象传入，LauncherAppState 的 setLauncher 方法如下所示：

packages/apps/Launcher3/src/com/android/launcher3/LauncherAppState.java

```

LauncherModel setLauncher(Launcher launcher) {
    getLocalProvider(mContext).setLauncherProviderChangeListener(launcher);
    mModel.initialize(launcher); //1
    return mModel;
}

```

在注释 1 处会调用 LauncherModel 的 initialize 方法：

packages/apps/Launcher3/src/com/android/launcher3/LauncherModel.java

```

public void initialize(Callbacks callbacks) {
    synchronized (mLock) {
        unbindItemInfosAndClearQueuedBindRunnables();
        mCallbacks = new WeakReference<Callbacks>(callbacks);
    }
}

```

在 initialize 方法中会将 Callbacks，也就是传入的 Launcher，封装成一个弱引用对象。因此我们得知 mCallbacks 变量指的就是封装成弱引用对象的 Launcher，这个 mCallbacks③后面会用到它。

再回到 Launcher 的 onCreate 方法，在注释 3 处调用了 LauncherModel 的 startLoader 方法，如下所示：

packages/apps/Launcher3/src/com/android/launcher3/LauncherModel.java

```

...
// 创建了具有消息循环的线程 HandlerThread 对象
@Thunk static final HandlerThread sWorkerThread = new HandlerThread("launcher-
loader"); //1
static {
    sWorkerThread.start();
}
@Thunk static final Handler sWorker = new Handler(sWorkerThread.getLooper()); //2

```

```

...
public void startLoader(boolean isLaunching, int synchronousBindPage) {
    synchronized (mLock) {
        if (DEBUG_LOADERS) {
            Log.d(TAG, "startLoader isLaunching=" + isLaunching);
        }
        mDeferredBindRunnables.clear();
        if (mCallbacks != null && mCallbacks.get() != null) {
            isLaunching = isLaunching || stopLoaderLocked();
            mLoaderTask = new LoaderTask(mApp, isLaunching);//3
            if (synchronousBindPage > -1 && mAllAppsLoaded && mWorkspaceLoaded) {
                mLoaderTask.runBindSynchronousPage(synchronousBindPage);
            } else {
                sWorkerThread.setPriority(Thread.NORM_PRIORITY);
                sWorker.post(mLoaderTask);//4
            }
        }
    }
}
}

```

在注释1处创建了具有消息循环的线程 HandlerThread 对象。在注释2处创建了 Handler，并且传入 HandlerThread 的 Looper，这里 Handler 的作用就是向 HandlerThread 发送消息。在注释3处创建 LoaderTask，在注释4处将 LoaderTask 作为消息发送给 HandlerThread。

LoaderTask 类实现了 Runnable 接口，当 LoaderTask 所描述的消息被处理时，则会调用它的 run 方法，LoaderTask 是 LauncherModel 的内部类，代码如下所示：

```

packages/apps/Launcher3/src/com/android/launcher3/LauncherModel.java

private class LoaderTask implements Runnable {
    ...
    public void run() {
        synchronized (mLock) {
            if (mStopped) {
                return;
            }
            mIsLoaderTaskRunning = true;
        }
        try {
            if (DEBUG_LOADERS) Log.d(TAG, "step 1.1: loading workspace");
            mIsLoadingAndBindingWorkspace = true;
            //加载工作区信息
            loadWorkspace();//1
            verifyNotStopped();
            if (DEBUG_LOADERS) Log.d(TAG, "step 1.2: bind workspace workspace");

```

```

        //绑定工作区信息
        bindWorkspace(mPageToBindFirst);//2
        if (DEBUG_LOADERS) Log.d(TAG, "step 1 completed, wait for idle");
        waitForIdle();
        verifyNotStopped();
        if (DEBUG_LOADERS) Log.d(TAG, "step 2.1: loading all apps");
        //加载系统已经安装的应用程序信息
        loadAllApps();//3
        ...
    } catch (CancellationException e) {
    } finally {
        ...
    }
}
...
}

```

Launcher 是用工作区的形式来显示系统安装的应用程序的快捷图标，每一个工作区都是用来描述一个抽象桌面的，它由 n 个屏幕组成，每个屏幕又分为 n 个单元格，每个单元格用来显示一个应用程序的快捷图标。在注释 1 处和注释 2 处分别调用 `loadWorkspace` 方法和 `bindWorkspace` 方法来加载和绑定工作区信息。注释 3 处的 `loadAllApps` 方法用来加载系统已经安装的应用程序信息，代码如下所示：

packages/apps/Launcher3/src/com/android/launcher3/LauncherModel.java

```

private void loadAllApps() {
    ...
    mHandler.post(new Runnable() {
        public void run() {
            final long bindTime = SystemClock.uptimeMillis();
            final Callbacks callbacks = tryGetCallbacks(oldCallbacks);
            if (callbacks != null) {
                callbacks.bindAllApplications(added);//1
                if (DEBUG_LOADERS) {
                    Log.d(TAG, "bound " + added.size() + " apps in "
                        + (SystemClock.uptimeMillis() - bindTime) + "ms");
                }
            } else {
                Log.i(TAG, "not binding apps: no Launcher activity");
            }
        }
    });
    ...
}

```

在注释 1 处会调用 `callbacks` 的 `bindAllApplications` 方法，从之前的标注③处我们得

知这个 callbacks 实际是指向 Launcher 的,下面我们来查看 Launcher 的 bindAllApplications 方法:

```
packages/apps/Launcher3/src/com/android/launcher3/Launcher.java

public void bindAllApplications(final ArrayList<AppInfo> apps) {
    if (waitUntilResume(mBindAllApplicationsRunnable, true)) {
        mTmpAppsList = apps;
        return;
    }
    if (mAppsView != null) {
        mAppsView.setApps(apps); //1
    }
    if (mLauncherCallbacks != null) {
        mLauncherCallbacks.bindAllApplications(apps);
    }
}
```

在注释 1 处会调用 AllAppsContainerView 类型的 mAppsView 对象的 setApps 方法,并将包含应用信息的列表 apps 传进去, AllAppsContainerView 的 setApps 方法如下所示:

```
packages/apps/Launcher3/src/com/android/launcher3/allapps/AllAppsContainerView.java

public void setApps(List<AppInfo> apps) {
    mApps.setApps(apps);
}
```

setApps 方法会将包含应用信息的列表 apps 设置给 mApps, 这个 mApps 是 AlphabeticalAppsList 类型的对象。接着查看 AllAppsContainerView 的 onFinishInflate 方法如下所示:

```
packages/apps/Launcher3/src/com/android/launcher3/allapps/AllAppsContainerView.java

@Override
protected void onFinishInflate() {
    super.onFinishInflate();
    ...
    mAppsRecyclerView = (AllAppsRecyclerView) findViewById(R.id.apps_list_
view); //1
    mAppsRecyclerView.setApps(mApps); //2
    mAppsRecyclerView.setLayoutManager(mLayoutManager);
    mAppsRecyclerView.setAdapter(mAdapter); //3
    ...
}
```

onFinishInflate 方法会在 AllAppsContainerView 加载完 XML 布局时调用,在注释 1 处得到 AllAppsRecyclerView 用来显示 App 列表,并在注释 2 处将此前的 mApps 设置进去,

在注释 3 处为 AllAppsRecyclerView 设置 Adapter。这样显示应用程序快捷图标的列表就会显示在屏幕上。

到这里 Launcher 中应用图标显示过程以及 Launcher 启动流程就讲解完了, 接下来介绍 Android 系统启动流程。

2.5 Android 系统启动流程

结合本章前 4 节的内容, 我们可以清晰地总结出 Android 系统启动流程, 这个流程主要有以下几个部分。

1. 启动电源以及系统启动

当电源按下时引导芯片代码从预定义的地方 (固化在 ROM) 开始执行。加载引导程序 BootLoader 到 RAM, 然后执行。

2. 引导程序 BootLoader

引导程序 BootLoader 是在 Android 操作系统开始运行前的一个小程序, 它的主要作用是把系统 OS 拉起来并运行。

3. Linux 内核启动

当内核启动时, 设置缓存、被保护存储器、计划列表、加载驱动。当内核完成系统设置时, 它首先在系统文件中寻找 init.rc 文件, 并启动 init 进程。

4. init 进程启动

初始化和启动属性服务, 并且启动 Zygote 进程。

5. Zygote 进程启动

创建 Java 虚拟机并为 Java 虚拟机注册 JNI 方法, 创建服务器端 Socket, 启动 SystemServer 进程。

6. SystemServer 进程启动

启动 Binder 线程池和 SystemServiceManager, 并且启动各种系统服务。

7. Launcher 启动

被 SystemServer 进程启动的 AMS 会启动 Launcher，Launcher 启动后会将已安装应用的快捷图标显示到界面上。

结合上面的流程，给出 Android 系统启动流程图，如图 2-4 所示。

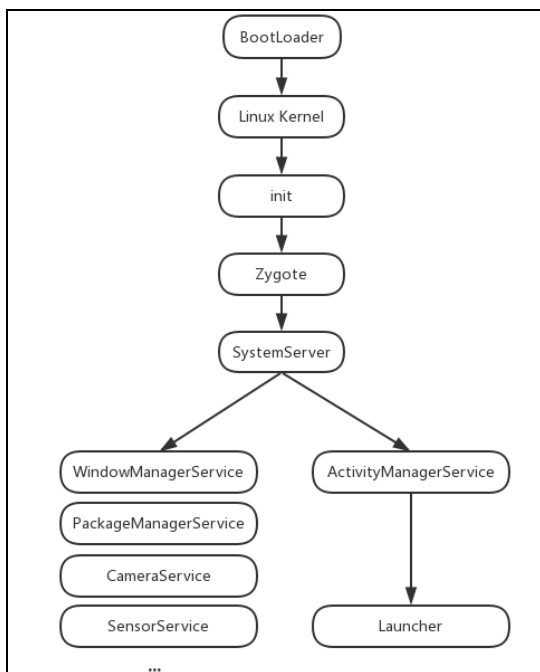


图 2-4 Android 系统启动流程图

实际上 Android 系统启动流程要比图 2-4 复杂得多，这里只是为了便于理解简化了很多细节。对于 Android 应用开发来说，这些知识点已经足够了。

2.6 本章小结

为了更好地理解 Android 系统启动流程，本章中我们先后学习了 init 进程启动过程、Zygote 进程启动过程、SystemServer 进程处理过程和 Launcher 启动过程，这些进程的启动过程其实细节很多也很复杂，而本章更注重流程，因此并不会对每一个细节进行深挖，如果想要深挖就需要读者自行阅读源码。另外，本章所讲到的知识会和后面章节有所关联，是全书的基础章节。

第 3 章

应用程序进程启动过程

关联章节：第 2 章 Android 系统启动

第 2 章我们学习了 Android 系统的启动流程，系统启动后，我们比较关心应用程序是如何启动的，启动一个应用程序首先要保证该应用程序的进程已经被启动，本章我们就来学习应用程序进程启动过程。需要注意，是“应用程序进程启动过程”，不是“应用程序启动过程”，关于应用程序启动过程（根 Activity 启动过程）将在第 4 章进行讲解。

3.1 应用程序进程简介

要想启动一个应用程序，首先要保证这个应用程序所需要的应用程序进程已经启动。AMS 在启动应用程序时会检查这个应用程序需要的应用程序进程是否存在，不存在就会请求 Zygote 进程启动需要的应用程序进程。在 2.2 节中，我们知道在 Zygote 的 Java 框架层中会创建一个 Server 端的 Socket，这个 Socket 用来等待 AMS 请求 Zygote 来创建新的应用程序进程。Zygote 进程通过 fork 自身创建应用程序进程，这样应用程序进程就会获得 Zygote 进程在启动时创建的虚拟机实例。当然，在应用程序进程创建过程中除了获取虚拟机实例外，还创建了 Binder 线程池和消息循环，这样运行在应用进程中的应用程序就可以方便地使用 Binder 进行进程间通信以及处理消息了。

3.2 应用程序进程启动过程介绍

应用程序进程创建过程的步骤比较多，这里分为两个部分来讲解，分别是 AMS 发送启动应用程序进程请求，以及 Zygote 接收请求并创建应用程序进程。

3.2.1 AMS发送启动应用程序进程请求

这里先给出 AMS 发送启动应用程序进程请求过程的时序图，然后对每一个步骤进行详细分析，如图 3-1 所示。

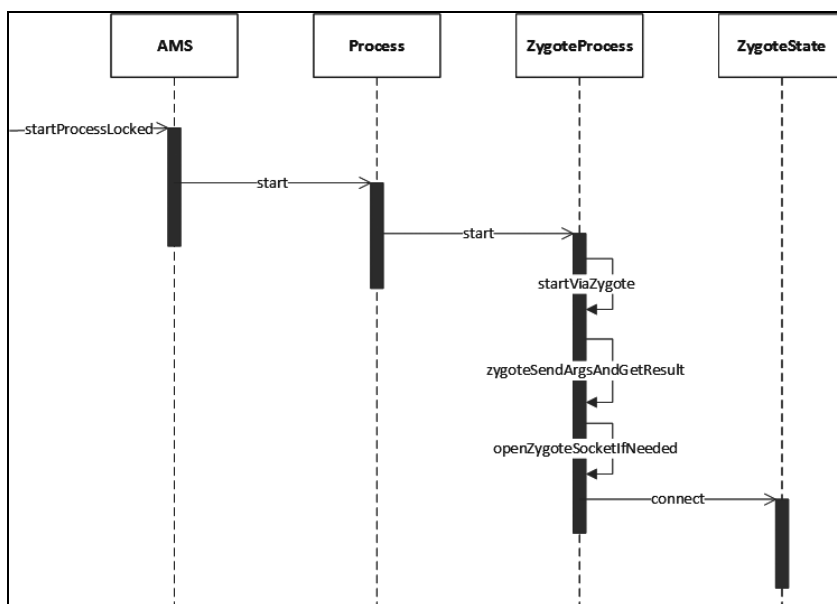


图 3-1 AMS 发送启动应用程序进程请求过程的时序图

AMS 如果想要启动应用程序进程，就需要向 Zygote 进程发送创建应用程序进程的请求，AMS 会通过调用 startProcessLocked 方法向 Zygote 进程发送请求，如下所示：

frameworks/base/services/core/java/com/android/server/am/ActivityManagerService.java

```

private final void startProcessLocked(ProcessRecord app, String hostingType,
    String hostingNameStr, String abiOverride, String entryPoint, String[]
    entryPointArgs) {
    ...
    try {
        try {
            final int userId = UserHandle.getUserId(app.uid);

```

```

        AppGlobals.getPackageManager().checkPackageStartable(app.info.
            packageName, userId);
    } catch (RemoteException e) {
        throw e.rethrowAsRuntimeException();
    }
}
//获取要创建的应用程序进程的用户 ID
int uid = app.uid;//1
int[] gids = null;
int mountExternal = Zygote.MOUNT_EXTERNAL_NONE;
if (!app.isolated) {
    ...
    /**
     * 2 对 gids 进行创建和赋值
     */
    if (ArrayUtils.isEmpty(permGids)) {
        gids = new int[3];
    } else {
        gids = new int[permGids.length + 3];
        System.arraycopy(permGids, 0, gids, 3, permGids.length);
    }
    gids[0] = UserHandle.getSharedAppGid(UserHandle.getAppId(uid));
    gids[1] = UserHandle.getCacheAppGid(UserHandle.getAppId(uid));
    gids[2] = UserHandle.getUserGid(UserHandle.getUserId(uid));
}
...
if (entryPoint == null) entryPoint = "android.app.ActivityThread";//3
Trace.traceBegin(Trace.TRACE_TAG_ACTIVITY_MANAGER, "Start proc: " +
    app.processName);
checkTime(startTime, "startProcess: asking zygote to start proc");
ProcessStartResult startResult;
if (hostingType.equals("webview_service")) {
    startResult = startWebView(entryPoint,
        app.processName, uid, uid, gids, debugFlags, mountExternal,
        app.info.targetSdkVersion, seInfo, requiredAbi, instructionSet,
        app.info.dataDir, null, entryPointArgs);
} else {
    /**
     * 4 启动应用程序进程
     */
    startResult = Process.start(entryPoint,
        app.processName, uid, uid, gids, debugFlags, mountExternal,
        app.info.targetSdkVersion, seInfo, requiredAbi, instructionSet,
        app.info.dataDir, invokeWith, entryPointArgs);
}
}

```

```

    ...
    } catch (RuntimeException e) {
        ...
    }
}

```

在注释 1 处得到创建应用程序进程的用户 ID，在注释 2 处对用户组 ID (gids) 进行创建和赋值。在注释 3 处如果 `entryPoint` 为 `null`，则赋值为 `android.app.ActivityThread`，这个值就是应用程序进程主线程的类名。在注释 4 处调用 `Process` 的 `start` 方法，将此前得到的应用程序进程用户 ID 和用户组 ID 传进去，第一个参数 `entryPoint` 我们得知是 `android.app.ActivityThread`，后面章节还会介绍它。接下来查看 `Process` 的 `start` 方法，如下所示：

frameworks/base/core/java/android/os/Process.java

```

public static final ProcessStartResult start(final String processClass,
                                             final String niceName,
                                             int uid, int gid, int[] gids,
                                             int debugFlags, int mountExternal,
                                             int targetSdkVersion,
                                             String seInfo,
                                             String abi,
                                             String instructionSet,
                                             String appDataDir,
                                             String invokeWith,
                                             String[] zygoteArgs) {
    return zygoteProcess.start(processClass, niceName, uid, gid, gids,
                               debugFlags, mountExternal, targetSdkVersion, seInfo,
                               abi, instructionSet, appDataDir, invokeWith, zygoteArgs);
}

```

在 `Process` 的 `start` 方法中只调用了 `ZygoteProcess` 的 `start` 方法，其中 `ZygoteProcess` 类用于保持与 `Zygote` 进程的通信状态。该 `start` 方法如下所示：

frameworks/base/core/java/android/os/ZygoteProcess.java

```

public final Process.ProcessStartResult start(final String processClass,
                                             final String niceName,
                                             int uid, int gid, int[] gids,
                                             int debugFlags, int mountExternal,
                                             int targetSdkVersion,
                                             String seInfo,
                                             String abi,
                                             String instructionSet,
                                             String appDataDir,

```

```

        String invokeWith,
        String[] zygoteArgs) {
    try {
        return startViaZygote(processClass, niceName, uid, gid, gids,
            debugFlags, mountExternal, targetSdkVersion, seInfo,
            abi, instructionSet, appDataDir, invokeWith, zygoteArgs);
    } catch (ZygoteStartFailedEx ex) {
        Log.e(LOG_TAG,
            "Starting VM process through Zygote failed");
        throw new RuntimeException(
            "Starting VM process through Zygote failed", ex);
    }
}

```

ZygoteProcess 的 start 方法调用了 startViaZygote 方法，如下所示：

frameworks/base/core/java/android/os/ZygoteProcess.java

```

private Process.ProcessStartResult startViaZygote(final String processClass,
final String niceName, final int uid, final int gid, final int[] gids, int debugFlags,
int mountExternal, int targetSdkVersion, String seInfo, String abi,
String instructionSet, String appDataDir, String invokeWith,
String[] extraArgs) throws ZygoteStartFailedEx {
    /**
     * 1 创建字符串列表 argsForZygote，并将应用进程的启动参数保存在 argsForZygote 中
     */
    ArrayList<String> argsForZygote = new ArrayList<String>();
    argsForZygote.add("--runtime-args");
    argsForZygote.add("--setuid=" + uid);
    argsForZygote.add("--setgid=" + gid);
    if ((debugFlags & Zygote.DEBUG_ENABLE_JNI_LOGGING) != 0) {
        argsForZygote.add("--enable-jni-logging");
    }
    ...
    synchronized(mLock) {
        return zygoteSendArgsAndGetResult(openZygoteSocketIfNeeded(abi),
            argsForZygote);
    }
}

```

在注释 1 处创建了字符串列表 argsForZygote，并将启动应用进程的启动参数保存在 argsForZygote 中，方法的最后会调用 zygoteSendArgsAndGetResult 方法，需要注意的是，zygoteSendArgsAndGetResult 方法的第一个参数中调用了 openZygoteSocketIfNeeded 方法^①，而第二个参数是保存应用进程的启动参数的 argsForZygote。zygoteSendArgsAndGetResult 方法如下所示：

frameworks/base/core/java/android/os/ZygoteProcess.java

```

@GuardedBy("mLock")
private static Process.ProcessStartResult zygoteSendArgsAndGetResult(
    ZygoteState zygoteState, ArrayList<String> args)
    throws ZygoteStartFailedEx {
    try {
        int sz = args.size();
        for (int i = 0; i < sz; i++) {
            if (args.get(i).indexOf('\n') >= 0) {
                throw new ZygoteStartFailedEx("embedded newlines not allowed");
            }
        }
        final BufferedWriter writer = zygoteState.writer;
        final DataInputStream inputStream = zygoteState.inputStream;
        writer.write(Integer.toString(args.size()));
        writer.newLine();
        for (int i = 0; i < sz; i++) {
            String arg = args.get(i);
            writer.write(arg);
            writer.newLine();
        }
        writer.flush();
        Process.ProcessStartResult result = new Process.ProcessStartResult();
        result.pid = inputStream.readInt();
        result.useWrapper = inputStream.readBoolean();

        if (result.pid < 0) {
            throw new ZygoteStartFailedEx("fork() failed");
        }
        return result;
    } catch (IOException ex) {
        zygoteState.close();
        throw new ZygoteStartFailedEx(ex);
    }
}

```

`zygoteSendArgsAndGetResult` 方法的主要作用就是将传入的应用进程的启动参数 `argsForZygote` 写入 `ZygoteState` 中，`ZygoteState` 是 `ZygoteProcess` 的静态内部类，用于表示与 `Zygote` 进程通信的状态。结合前面的标注①我们知道 `ZygoteState` 其实是由 `openZygoteSocketIfNeeded` 方法返回的，那么我们接着来看 `openZygoteSocketIfNeeded` 方法做了什么，代码如下所示：


```
frameworks/base/core/java/android/os/ZygoteProcess.java
```

```
@GuardedBy("mLock")
private ZygoteState openZygoteSocketIfNeeded(String abi) throws ZygoteStartFailedEx {
    Preconditions.checkNotNull(Thread.holdsLock(mLock), "ZygoteProcess lock not held");

    if (primaryZygoteState == null || primaryZygoteState.isClosed()) {
        try {
            //与 Zygote 进程建立 Socket 连接
            primaryZygoteState = ZygoteState.connect(mSocket);//1
        } catch (IOException ioe) {
            throw new ZygoteStartFailedEx("Error connecting to primary zygote", ioe);
        }
    }
    //连接 Zygote 主模式返回的 ZygoteState 是否与启动应用程序进程所需要的 ABI 匹配
    if (primaryZygoteState.matches(abi)) {//2
        return primaryZygoteState;
    }

    //如果不匹配，则尝试连接 Zygote 辅模式
    if (secondaryZygoteState == null || secondaryZygoteState.isClosed()) {
        try {
            secondaryZygoteState = ZygoteState.connect(mSecondarySocket);//3
        } catch (IOException ioe) {
            throw new ZygoteStartFailedEx("Error connecting to secondary zygote", ioe);
        }
    }
    //连接 Zygote 辅模式返回的 ZygoteState 是否与启动应用程序进程所需要的 ABI 匹配
    if (secondaryZygoteState.matches(abi)) {//4
        return secondaryZygoteState;
    }
    throw new ZygoteStartFailedEx("Unsupported zygote ABI: " + abi);
}
```

在 2.2 节讲到 Zygote 进程启动过程时我们得知，在 Zygote 的 main 方法中会创建 name 为“zygote”的 Server 端 Socket。在注释 1 处会调用 ZygoteState 的 connect 方法与名称为 ZYGOTE_SOCKET 的 Socket 建立连接，这里 ZYGOTE_SOCKET 的值为“zygote”，也就是说，在注释 1 处与 Zygote 进程建立 Socket 连接，并返回 ZygoteState 类型的 primaryZygoteState 对象，在注释 2 处如果 primaryZygoteState 与启动应用程序进程所需的 ABI 不匹配，则会在注释 3 处连接 name 为“zygote_secondary”的 Socket。在 2.2.2 节中讲到过 Zygote 的启动脚本有 4 种，如果采用的是 init.zygote32_64.rc 或者 init.zygote64_32.rc，则 name 为“zygote”的为主模式，name 为“zygote_secondary”的为辅模式，那么注释 2

和注释 3 处的意思简单来说就是,如果连接 Zygote 主模式返回的 ZygoteState 与启动应用程序进程所需的 ABI 不匹配,则连接 Zygote 辅模式。如果在注释 4 处连接 Zygote 辅模式返回的 ZygoteState 与启动应用程序进程所需的 ABI 也不匹配,则抛出 ZygoteStartFailedEx 异常。

3.2.2 Zygote接收请求并创建应用程序进程

Zygote 接收请求并创建应用程序进程的时序图如图 3-2 所示。

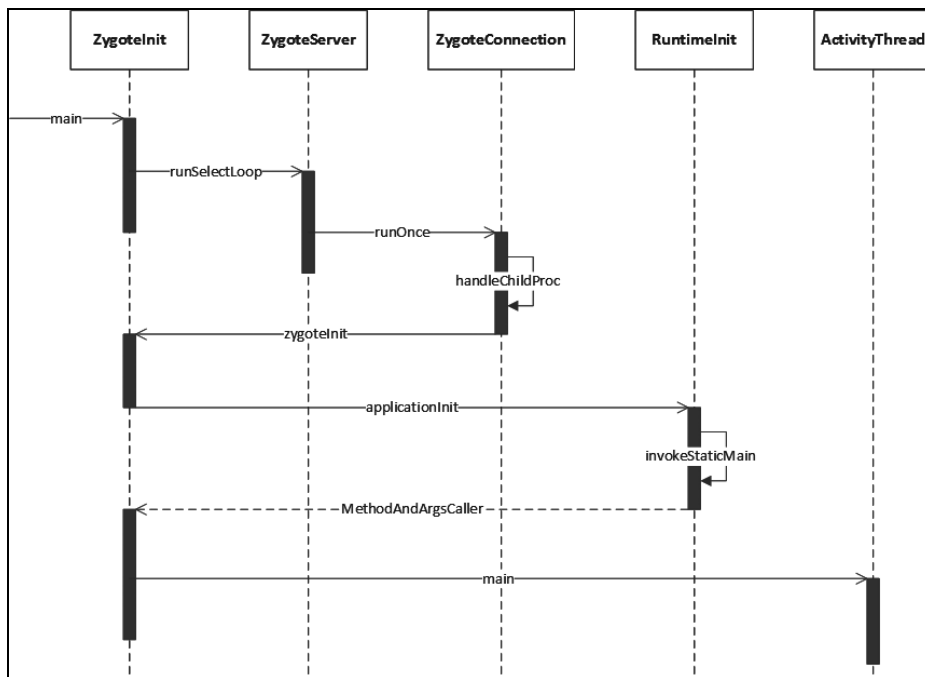


图 3-2 Zygote 接收请求并创建应用程序进程的时序图

Socket 连接成功并匹配 ABI 后会返回 ZygoteState 类型对象,我们在分析 zygoteSendArgsAndGetResult 方法中讲过,会将应用进程的启动参数 argsForZygote 写入 ZygoteState 中,这样 Zygote 进程就会收到一个创建新的应用程序进程的请求,我们回到 ZygoteInit 的 main 方法,如下所示:

```
frameworks/base/core/java/com/android/internal/os/ZygoteInit.java
```

```
public static void main(String argv[]) {
    ...
    try {
```

```

...
//创建一个 Server 端的 Socket, socketName 的值为 “zygote”
zygoteServer.registerServerSocket(socketName);//1
if (!enableLazyPreload) {
    bootTimingsTraceLog.traceBegin("ZygotePreload");
    EventLog.writeEvent(LOG_BOOT_PROGRESS_PRELOAD_START,
        SystemClock.uptimeMillis());
    //预加载类和资源
    preload(bootTimingsTraceLog);//2
    EventLog.writeEvent(LOG_BOOT_PROGRESS_PRELOAD_END,
        SystemClock.uptimeMillis());
    bootTimingsTraceLog.traceEnd();
} else {
    Zygote.resetNicePriority();
}
...
if (startSystemServer) {
    //启动 SystemServer 进程
    startSystemServer(abiList, socketName, zygoteServer);//3
}
Log.i(TAG, "Accepting command socket connections");
//等待 AMS 请求
zygoteServer.runSelectLoop(abiList);//4
zygoteServer.closeServerSocket();
} catch (Zygote.MethodAndArgsCaller caller) {
    caller.run();
} catch (Throwable ex) {
    Log.e(TAG, "System zygote died with exception", ex);
    zygoteServer.closeServerSocket();
    throw ex;
}
}

```

这些内容在 2.2.3 节中讲过,但为了更好地理解本节内容,这里再讲一遍。在注释 1 处通过 registerZygoteSocket 方法创建了一个 Server 端的 Socket,这个 name 为 “zygote” 的 Socket 用来等待 AMS 请求 Zygote,以创建新的应用程序进程,关于 AMS 后面的章节会进行介绍。在注释 2 处预加载类和资源。在注释 3 处启动 SystemServer 进程,这样系统的服务也会由 SystemServer 进程启动起来。在注释 4 处调用 ZygoteServer 的 runSelectLoop 方法来等待 AMS 请求创建新的应用程序进程。下面来查看 ZygoteServer 的 runSelectLoop 方法:

frameworks/base/core/java/com/android/internal/os/ZygoteServer.java

```

void runSelectLoop(String abiList) throws Zygote.MethodAndArgsCaller {
    ArrayList<FileDescriptor> fds = new ArrayList<FileDescriptor>();

```

```

        ArrayList<ZygoteConnection> peers = new ArrayList<ZygoteConnection>();//1
        fds.add(mServerSocket.getFileDescriptor());
        peers.add(null);
        while (true) {
            ...
            for (int i = pollFds.length - 1; i >= 0; --i) {
                if ((pollFds[i].revents & POLLIN) == 0) {
                    continue;
                }
                if (i == 0) {
                    ZygoteConnection newPeer = acceptCommandPeer(abiList);
                    peers.add(newPeer);
                    fds.add(newPeer.getFileDescriptor());
                } else {
                    boolean done = peers.get(i).runOnce(this);//2
                    if (done) {
                        peers.remove(i);
                        fds.remove(i);
                    }
                }
            }
        }
    }
}

```

当有 AMS 的请求数据到来时，会调用注释 2 处的代码，结合注释 1 处的代码，我们得知注释 2 处的代码其实是调用 ZygoteConnection 的 runOnce 方法来处理请求数据的：

frameworks/base/core/java/com/android/internal/os/ZygoteConnection.java

```

boolean runOnce(ZygoteServer zygoteServer) throws Zygote.MethodAndArgsCaller {
    String args[];
    Arguments parsedArgs = null;
    FileDescriptor[] descriptors;
    try {
        //获取应用程序进程的启动参数
        args = readArgumentList();//1
        descriptors = mSocket.getAncillaryFileDescriptors();
    } catch (IOException ex) {
        Log.w(TAG, "IOException on command socket " + ex.getMessage());
        closeSocket();
        return true;
    }
    ...
    try {

```

```

        parsedArgs = new Arguments(args);//2
    ...
    /**
     * 3 创建应用程序进程
     */
    pid = Zygote.forkAndSpecialize(parsedArgs.uid, parsedArgs.gid, parsedArgs.gids,
        parsedArgs.debugFlags, rlimits, parsedArgs.mountExternal, parsedArgs.seInfo,
        parsedArgs.niceName, fdsToClose, fdsToIgnore, parsedArgs.instructionSet,
        parsedArgs.appDataDir);
} catch (ErrnoException ex) {
    ...
}
try {
    //当前代码逻辑运行在子进程中
    if (pid == 0) {
        zygoteServer.closeServerSocket();
        IoUtils.closeQuietly(serverPipeFd);
        serverPipeFd = null;
        //处理应用程序进程
        handleChildProc(parsedArgs, descriptors, childPipeFd, newStderr);
        return true;
    } else {
        IoUtils.closeQuietly(childPipeFd);
        childPipeFd = null;
        return handleParentProc(pid, descriptors, serverPipeFd, parsedArgs);
    }
} finally {
    IoUtils.closeQuietly(childPipeFd);
    IoUtils.closeQuietly(serverPipeFd);
}
}
}

```

在注释 1 处调用 `readArgumentList` 方法来获取应用程序进程的启动参数，并在注释 2 处将 `readArgumentList` 方法返回的字符串数组 `args` 封装到 `Arguments` 类型的 `parsedArgs` 对象中。在注释 3 处调用 `Zygote` 的 `forkAndSpecialize` 方法来创建应用程序进程，参数为 `parsedArgs` 中存储的应用进程启动参数，返回值为 `pid`。`forkAndSpecialize` 方法主要是通过 `fork` 当前进程来创建一个子进程的，如果 `pid` 等于 0，则说明当前代码逻辑运行在新创建的子进程（应用程序进程）中，这时就会调用 `handleChildProc` 方法来处理应用程序进程，如下所示：

```
frameworks/base/core/java/com/android/internal/os/ZygoteConnection.java
```

```
private void handleChildProc(Arguments parsedArgs,
```

```

        FileDescriptor[] descriptors, FileDescriptor pipeFd, PrintStream
        newStderr)
        throws Zygote.MethodAndArgsCaller {
    ...
    if (parsedArgs.invokeWith != null) {
        WrapperInit.execApplication(parsedArgs.invokeWith,
            parsedArgs.niceName, parsedArgs.targetSdkVersion,
            VMRuntime.getCurrentInstructionSet(),
            pipeFd, parsedArgs.remainingArgs);
    } else {
        ZygoteInit.zygoteInit(parsedArgs.targetSdkVersion,
            parsedArgs.remainingArgs, null /* classLoader */);
    }
}

```

handleChildProc 方法中调用了 ZygoteInit 的 zygoteInit 方法，如下所示：

frameworks/base/core/java/com/android/internal/os/ZygoteInit.java

```

public static final void zygoteInit(int targetSdkVersion, String[] argv,
    ClassLoader classLoader) throws Zygote.MethodAndArgsCaller {
    if (RuntimeInit.DEBUG) {
        Slog.d(RuntimeInit.TAG, "RuntimeInit: Starting application from zygote");
    }
    Trace.traceBegin(Trace.TRACE_TAG_ACTIVITY_MANAGER, "ZygoteInit");
    RuntimeInit.redirectLogStreams();
    RuntimeInit.commonInit();
    ZygoteInit.nativeZygoteInit();//1
    RuntimeInit.applicationInit(targetSdkVersion, argv, classLoader);//2
}

```

在注释 1 处会在新创建的应用程序进程中创建 Binder 线程池，这将在 3.3 节详细介绍。
在注释 2 处调用了 RuntimeInit 的 applicationInit 方法：

frameworks/base/core/java/com/android/internal/os/RuntimeInit.java

```

protected static void applicationInit(int targetSdkVersion, String[] argv,
    ClassLoader classLoader)
    throws Zygote.MethodAndArgsCaller {
    ...
    final Arguments args;
    try {
        args = new Arguments(argv);
    } catch (IllegalArgumentException ex) {
        Slog.e(TAG, ex.getMessage());
        return;
    }
}

```

```

}
Trace.traceEnd(Trace.TRACE_TAG_ACTIVITY_MANAGER);
invokeStaticMain(args.startClass, args.startArgs, classLoader);//1

```

在 `applicationInit` 方法中会在注释 1 处调用 `invokeStaticMain` 方法，需要注意的是，第一个参数 `args.startClass`，它指的就是本章开头提到的参数 `android.app.ActivityThread`。接下来我们查看 `invokeStaticMain` 方法，如下所示：

`frameworks/base/core/java/com/android/internal/os/RuntimeInit.java`

```

private static void invokeStaticMain(String className, String[] argv, ClassLoader
classLoader)
    throws Zygote.MethodAndArgsCaller {
    Class<?> cl;
    try {
        //获得 android.app.ActivityThread 类
        cl = Class.forName(className, true, classLoader);//1
    } catch (ClassNotFoundException ex) {
        throw new RuntimeException(
            "Missing class when invoking static main " + className,
            ex);
    }
    Method m;
    try {
        //获得 ActivityThread 的 main 方法
        m = cl.getMethod("main", new Class[] { String[].class });//2
    } catch (NoSuchMethodException ex) {
        throw new RuntimeException(
            "Missing static main on " + className, ex);
    }
    ...
    throw new Zygote.MethodAndArgsCaller(m, argv);//3
}

```

可以看到在注释 1 处通过反射获得了 `android.app.ActivityThread` 类，接下来在注释 2 处获得了 `ActivityThread` 的 `main` 方法，并将 `main` 方法传入注释 3 处的 `Zygote` 中的 `MethodAndArgsCaller` 类的构造方法中。在注释 3 处抛出的 `MethodAndArgsCaller` 异常会被 `Zygote` 的 `main` 方法捕获，至于这里为何采用了抛出异常而不是直接调用 `ActivityThread` 的 `main` 方法，原理和本书 2.3.1 节 `Zygote` 处理 `SystemServer` 进程是一样的，这种抛出异常的处理会清除所有的设置过程需要的堆栈帧，并让 `ActivityThread` 的 `main` 方法看起来像是应用程序进程的入口方法。下面来查看 `ZygoteInit.java` 的 `main` 方法是如何捕获 `MethodAndArgsCaller` 异常的，如下所示：

```
frameworks/base/core/java/com/android/internal/os/ZygoteInit.java
```

```
public static void main(String argv[]) {
    ...
    closeServerSocket();
} catch (MethodAndArgsCaller caller) {
    caller.run();//1
} catch (RuntimeException ex) {
    Log.e(TAG, "Zygote died with exception", ex);
    closeServerSocket();
    throw ex;
}
}
```

当捕获到 MethodAndArgsCaller 异常时,就会在注释 1 处调用 MethodAndArgsCaller 的 run 方法, MethodAndArgsCaller 是 Zygote.java 的静态内部类:

```
frameworks/base/core/java/com/android/internal/os/Zygote.java
```

```
public static class MethodAndArgsCaller extends Exception
    implements Runnable {
    private final Method mMethod;
    private final String[] mArgs;
    public MethodAndArgsCaller(Method method, String[] args) {
        mMethod = method;
        mArgs = args;
    }
    public void run() {
        try {
            mMethod.invoke(null, new Object[] { mArgs });//1
        } catch (IllegalAccessException ex) {
            throw new RuntimeException(ex);
        }
        ...
    }
}
```

注释 1 处的 mMethod 指的就是 ActivityThread 的 main 方法,调用了 mMethod 的 invoke 方法后,ActivityThread 的 main 方法就会被动态调用,应用程序进程就进入了 ActivityThread 的 main 方法中。讲到这里,应用程序进程就创建完成了并且运行了主线程的管理类 ActivityThread。

3.3 Binder线程池启动过程

在 3.2.2 节中学习了 Zygote 接收请求并创建应用程序进程，其中有一个遗留的知识点就是，在应用程序进程创建过程中会启动 Binder 线程池。我们查看 ZygoteInit 类的 zygoteInit 方法，如下所示：

```
frameworks/base/core/java/com/android/internal/os/ZygoteInit.java
```

```
public static final void zygoteInit(int targetSdkVersion, String[] argv,
    ClassLoader classLoader) throws Zygote.MethodAndArgsCaller {
    if (RuntimeInit.DEBUG) {
        Slog.d(RuntimeInit.TAG, "RuntimeInit: Starting application from zygote");
    }
    Trace.traceBegin(Trace.TRACE_TAG_ACTIVITY_MANAGER, "ZygoteInit");
    RuntimeInit.redirectLogStreams();
    RuntimeInit.commonInit();
    ZygoteInit.nativeZygoteInit();//1
    RuntimeInit.applicationInit(targetSdkVersion, argv, classLoader);
}
```

在注释 1 处会在新创建的应用程序进程中创建 Binder 线程池，下面来查看 nativeZygoteInit 方法：

```
frameworks/base/core/java/com/android/internal/os/ZygoteInit.java
```

```
private static final native void nativeZygoteInit();
```

很明显 nativeZygoteInit 是一个 JNI 方法，它对应的函数是什么呢？在 AndroidRuntime.cpp 的 JNINativeMethod 数组中我们得知它对应的函数是 com_android_internal_os_ZygoteInit_nativeZygoteInit，如下所示：

```
frameworks/base/core/jni/AndroidRuntime.cpp
```

```
const JNINativeMethod methods[] = {
    { "nativeZygoteInit", "()V",
      (void*) com_android_internal_os_ZygoteInit_nativeZygoteInit },
};
```

接着来查看 com_android_internal_os_ZygoteInit_nativeZygoteInit 函数：

```
frameworks/base/core/jni/AndroidRuntime.cpp
```

```
static void com_android_internal_os_ZygoteInit_nativeZygoteInit(JNIEnv* env, jobject
clazz)
{
    gCurRuntime->onZygoteInit();
}
```

`gCurRuntime` 是 `AndroidRuntime` 类型的指针，它是在 `AndroidRuntime` 初始化时就创建的，如下所示：

```
frameworks/base/core/jni/AndroidRuntime.cpp
```

```
...
static AndroidRuntime* gCurRuntime = NULL;
...
AndroidRuntime::AndroidRuntime(char* argBlockStart, const size_t argBlockLength) :
    mExitWithoutCleanup(false),
    mArgBlockStart(argBlockStart),
    mArgBlockLength(argBlockLength)
{
    ...
    gCurRuntime = this;
}
```

`AppRuntime` 继承自 `AndroidRuntime`，`AppRuntime` 创建时就会调用 `AndroidRuntime` 的构造函数，`gCurRuntime` 就会被初始化，它指向的是 `AppRuntime`，我们来查看 `AppRuntime` 的 `onZygoteInit` 函数，`AppRuntime` 在 `app_main.cpp` 中实现，如下所示：

```
frameworks/base/cmds/app_process/app_main.cpp
```

```
virtual void onZygoteInit()
{
    sp<ProcessState> proc = ProcessState::self();
    ALOGV("App process: starting thread pool.\n");
    proc->startThreadPool();
}
```

最后一行会调用 `ProcessState` 的 `startThreadPool` 函数来启动 Binder 线程池：

```
frameworks/native/libs/binder/ProcessState.cpp
```

```
void ProcessState::startThreadPool()
{
    AutoMutex _l(mLock);
    if (!mThreadPoolStarted) { //1
        mThreadPoolStarted = true; //2
        spawnPooledThread(true);
    }
}
```

支持 Binder 通信的进程中都有一个 `ProcessState` 类，它里面有一个 `mThreadPoolStarted` 变量，用来表示 Binder 线程池是否已经被启动过，默认值为 `false`。在每次调用 `startThreadPool`

函数时都会在注释 1 处先检查这个标记，从而确保 Binder 线程池只会被启动一次。如果 Binder 线程池未被启动，则在注释 2 处设置 `mThreadPoolStarted` 为 `true`，并调用 `spawnPooledThread` 函数来创建线程池中的第一个线程，也就是线程池的主线程，如下所示：

```
frameworks/native/libs/binder/ProcessState.cpp
```

```
void ProcessState::spawnPooledThread(bool isMain)
{
    if (mThreadPoolStarted) {
        String8 name = makeBinderThreadName();
        ALOGV("Spawning new pooled thread, name=%s\n", name.string());
        sp<Thread> t = new PoolThread(isMain);
        t->run(name.string()); //1
    }
}
```

可以看到 Binder 线程为一个 `PoolThread`。在注释 1 处调用 `PoolThread` 的 `run` 函数来启动一个新的线程。下面来查看 `PoolThread` 类做了什么：

```
frameworks/native/libs/binder/ProcessState.cpp
```

```
class PoolThread : public Thread
{
..
protected:
    virtual bool threadLoop()
    {
        IPCThreadState::self()->joinThreadPool(mIsMain); //1
        return false;
    }
    const bool mIsMain;
};
```

`PoolThread` 类继承了 `Thread` 类。在注释 1 处调用 `IPCThreadState` 的 `joinThreadPool` 函数，将当前线程注册到 Binder 驱动程序中，这样我们创建的线程就加入了 Binder 线程池中，新创建的应用程序进程就支持 Binder 进程间通信了，我们只需要创建当前进程的 Binder 对象，并将它注册到 `ServiceManager` 中就可以实现 Binder 进程间通信，而不必关心进程间是如何通过 Binder 进行通信的。

3.4 消息循环创建过程

在 3.2.2 节中学习了 Zygote 接收请求并创建应用程序进程，还有一个遗留的知识点就是，应用程序进程启动后会创建消息循环。首先我们回到 RuntimeInit 的 invokeStaticMain 方法，代码如下所示：

```
frameworks/base/core/java/com/android/internal/os/RuntimeInit.java
```

```
private static void invokeStaticMain(String className, String[] argv, ClassLoader
classLoader)
    throws ZygoteInit.MethodAndArgsCaller {
    Class<?> cl;
    ...
    throw new ZygoteInit.MethodAndArgsCaller(m, argv);
}
```

invokeStaticMain 方法在 3.2 节已经讲过，这里不再赘述，主要是看最后一行，会抛出一个 MethodAndArgsCaller 异常，这个异常会被 ZygoteInit 的 main 方法捕获，如下所示：

```
frameworks/base/core/java/com/android/internal/os/ZygoteInit.java
```

```
public static void main(String argv[]) {
    ...
    try {
        ...
    } catch (MethodAndArgsCaller caller) {
        caller.run();//1
    } catch (RuntimeException ex) {
        Log.e(TAG, "Zygote died with exception", ex);
        closeServerSocket();
        throw ex;
    }
}
```

在注释 1 处捕获到 MethodAndArgsCaller 时会执行 caller 的 run 方法，如下所示：

```
frameworks/base/core/java/com/android/internal/os/ZygoteInit.java
```

```
public static class MethodAndArgsCaller extends Exception
implements Runnable {
    private final Method mMethod;
    private final String[] mArgs;
    public MethodAndArgsCaller(Method method, String[] args) {
        mMethod = method;
        mArgs = args;
    }
}
```

```

    }
    public void run() {
        try {
            mMethod.invoke(null, new Object[] { mArgs }); //1
        } catch (IllegalAccessException ex) {
            throw new RuntimeException(ex);
        }
        ...
        throw new RuntimeException(ex);
    }
}

```

根据 3.2.2 节我们得知，mMethod 指的就是 ActivityThread 的 main 方法，mArgs 指的是应用程序进程的启动参数。在注释 1 处调用 ActivityThread 的 main 方法，代码如下所示：

frameworks/base/core/java/android/app/ActivityThread.java

```

public static void main(String[] args) {
    ...
    //创建主线程 Looper
    Looper.prepareMainLooper(); //1
    ActivityThread thread = new ActivityThread(); //2
    thread.attach(false);
    if (sMainThreadHandler == null) { //3
        //创建主线程 H 类
        sMainThreadHandler = thread.getHandler(); //4
    }
    if (false) {
        Looper.myLooper().setMessageLogging(new
            LogPrinter(Log.DEBUG, "ActivityThread"));
    }
    Trace.traceEnd(Trace.TRACE_TAG_ACTIVITY_MANAGER);
    //Looper 开始工作
    Looper.loop(); //5
    throw new RuntimeException("Main thread loop unexpectedly exited");
}

```

ActivityThread 类用于管理当前应用程序进程的主线程，在注释 1 处创建主线程的消息循环 Looper，在注释 2 处创建 ActivityThread。在注释 3 处判断 Handler 类型的 sMainThreadHandler 是否为 null，如果为 null 则在注释 4 处获取 H 类并赋值给 sMainThreadHandler，这个 H 类继承自 Handler，是 ActivityThread 的内部类，用于处理主线程的消息循环，在第 4 章、第 5 章我们将会经常提到它。在注释 5 处调用 Looper 的 loop

方法，使得 `Looper` 开始处理消息。可以看出，系统在应用程序进程启动完成后，就会创建一个消息循环，这样运行在应用程序进程中的应用程序可以方便地使用消息处理机制。

3.5 本章小结

本章的内容不多却十分重要，开发人员了解自己所开发的应用的进程是如何创建的是十分必要的。本章以第 2 章为基础，同时又是第 4 章的基础，起着承上启下的作用。在 3.3 和 3.4 节我们学习了 `Binder` 线程池和消息循环是如何创建的，它们是进程和线程间通信的重要手段，其中 3.4 节讲到的 `H` 类，更是第 4 章、第 5 章将会经常提到的类。

第 4 章

四大组件的工作过程

关联章节：第 2 章 Android 系统启动；第 3 章 应用程序进程启动过程

在前面的两章中我们学习了系统的启动过程和应用进程的启动过程，应用进程启动后接着就该启动应用程序了，也就是启动根 Activity。而 Activity 是四大组件之一，因此本章我们就来学习四大组件的工作过程。四大组件是应用开发最常接触的，包括 Activity、Service、BroadcastReceiver 和 ContentProvider。本章不会介绍四大组件的含义以及如何使用，而是更加深入地介绍它们的工作过程，比如 Service 的启动过程。比起前面两章，本章的内容更是资深工程师所必须掌握的知识点之一。本章内容以前面两章的内容（系统的启动过程和应用进程的启动过程）为基础，同时又和插件化技术有所关联，想要理解插件化的原理就必须了解四大组件的工作过程，但最主要的是本章内容是整个 Android 知识体系的核心内容之一，对于理解和掌握整个 Android 知识体系起着重大的作用。

本章和前面两章一样不会拘泥于源码细节，而是注重流程，正确的阅读“姿势”就是阅读前先要查看时序图了解大概的流程，再阅读具体的代码流程，看完代码流程后再回顾一下时序图。需要注意的是，本书的源码基于 Android 8.0，本章所讲的四大组件的工作过程会和 Android 7.0 以及之前的版本有些区别。

4.1 根Activity的启动过程

Activity 的启动过程分为两种，一种是根 Activity 的启动过程，另一种是普通 Activity 的启动过程。根 Activity 指的是应用程序启动的第一个 Activity，因此根 Activity 的启动过程一般情况下也可以理解为应用程序的启动过程。普通 Activity 指的是除应用程序启动的第一个 Activity 之外的其他 Activity。这里介绍的是根 Activity 的启动过程，它和普通 Activity 的启动过程是有重叠部分的，只不过根 Activity 的启动过程一般情况下指的就是应用程序的启动过程，更具有指导性意义。想要了解普通 Activity 的启动过程，读者可以参考根 Activity 的启动过程，自行去阅读源码。

根 Activity 的启动过程比较复杂，因此这里分为 3 个部分来讲，分别是 Launcher 请求 AMS 过程、AMS 到 ApplicationThread 的调用过程和 ActivityThread 启动 Activity。

4.1.1 Launcher请求AMS过程

在 2.4.3 节中讲过 Launcher 启动后会将已安装应用程序的快捷图标显示到桌面上，这些应用程序的快捷图标就是启动根 Activity 的入口，当我们点击某个应用程序的快捷图标时，就会通过 Launcher 请求 AMS 来启动该应用程序。Launcher 请求 AMS 的时序图如图 4-1 所示。

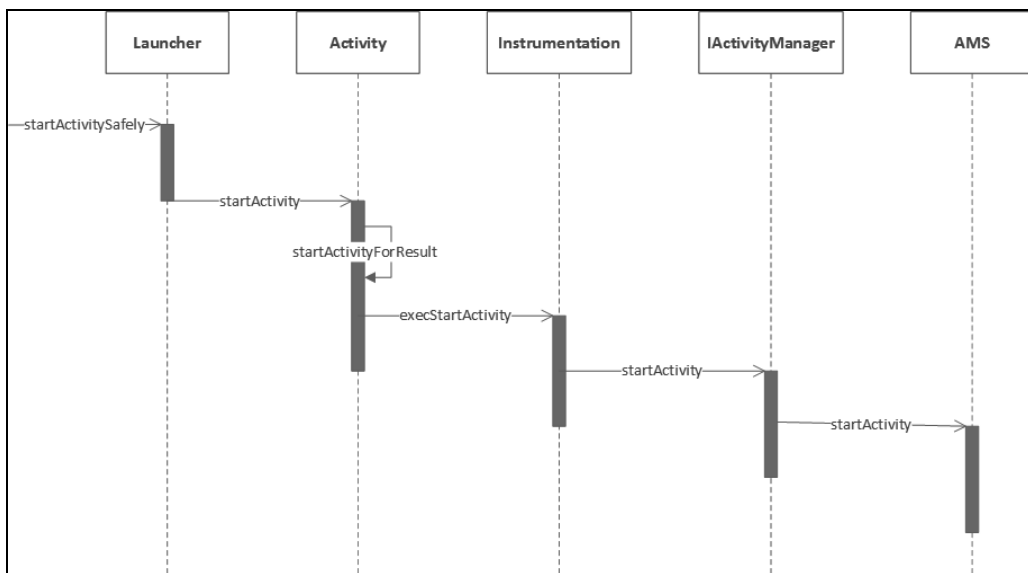


图 4-1 Launcher 请求 AMS 的时序图

当我们点击应用程序的快捷图标时，就会调用 Launcher 的 `startActivitySafely` 方法，如下所示：

`packages/apps/Launcher3/src/com/android/launcher3/Launcher.java`

```
public boolean startActivitySafely(View v, Intent intent, ItemInfo item) {
    ...
    intent.addFlags(Intent.FLAG_ACTIVITY_NEW_TASK); //1
    if (v != null) {
        intent.setSourceBounds(getViewBounds(v));
    }
    try {
        if (Utilities.ATLEAST_MARSHMALLOW
            && (item instanceof ShortcutInfo)
            && (item.itemType == Favorites.ITEM_TYPE_SHORTCUT
                || item.itemType == Favorites.ITEM_TYPE_DEEP_SHORTCUT)
            && !((ShortcutInfo) item).isPromise()) {
            startShortcutIntentSafely(intent, optsBundle, item);
        } else if (user == null || user.equals(Process.myUserHandle())) {
            startActivity(intent, optsBundle); //2
        } else {
            LauncherAppsCompat.getInstance(this).startActivityForProfile(
                intent.getComponent(), user, intent.getSourceBounds(), optsBundle);
        }
        return true;
    } catch (ActivityNotFoundException | SecurityException e) {
        Toast.makeText(this, R.string.activity_not_found, Toast.LENGTH_SHORT).
            show();
        Log.e(TAG, "Unable to launch. tag=" + item + " intent=" + intent, e);
    }
    return false;
}
```

在注释 1 处将 Flag 设置为 `Intent.FLAG_ACTIVITY_NEW_TASK`①，这样根 Activity 会在新的任务栈中启动。在注释 2 处会调用 `startActivity` 方法，这个 `startActivity` 方法在 Activity 中实现，如下所示：

`frameworks/base/core/java/android/app/Activity.java`

```
@Override
public void startActivity(Intent intent, @Nullable Bundle options) {
    if (options != null) {
        startActivityForResult(intent, -1, options);
    } else {
        startActivityForResult(intent, -1);
    }
}
```

在 `startActivity` 方法中会调用 `startActivityForResult` 方法，它的第二个参数为-1，表示 Launcher 不需要知道 Activity 启动的结果，`startActivityForResult` 方法的代码如下所示：

frameworks/base/core/java/android/app/Activity.java

```
public void startActivityForResult(@RequiresPermission Intent intent, int requestCode,
    @Nullable Bundle options) {
    if (mParent == null) { //1
        options = transferSpringboardActivityOptions(options);
        Instrumentation.ActivityResult ar =
            mInstrumentation.execStartActivity(
                this, mMainThread.getApplicationThread(), mToken, this,
                intent, requestCode, options);
        ...
    } else {
        ...
    }
}
```

注释 1 处的 `mParent` 是 Activity 类型的，表示当前 Activity 的父类。因为目前根 Activity 还没有创建出来，因此，`mParent == null` 成立。接着调用 `Instrumentation` 的 `execStartActivity` 方法，`Instrumentation` 主要用来监控应用程序和系统的交互，`execStartActivity` 方法的代码如下所示：

frameworks/base/core/java/android/app/Instrumentation.java

```
public ActivityResult execStartActivity(
    Context who, IBinder contextThread, IBinder token, Activity target,
    Intent intent, int requestCode, Bundle options) {
    ...
    try {
        intent.migrateExtraStreamToClipData();
        intent.prepareToLeaveProcess(who);
        int result = ActivityManager.getService()
            .startActivity(whoThread, who.getBasePackageName(), intent,
                intent.resolveTypeIfNeeded(who.getContentResolver()),
                token, target != null ? target.mEmbeddedID : null,
                requestCode, 0, null, options);
        checkStartActivityResult(result, intent);
    } catch (RemoteException e) {
        throw new RuntimeException("Failure from system", e);
    }
    return null;
}
```

首先调用 `ActivityManager` 的 `getService` 方法来获取 AMS 的代理对象，接着调用它的

startActivity 方法。这里与 Android 8.0 之前代码的逻辑有些不同，Android 8.0 之前是通过 ActivityManagerNative 的 getDefault 来获取 AMS 的代理对象的，现在这个逻辑封装到了 ActivityManager 中而不是 ActivityManagerNative 中。首先我们来查看 ActivityManager 的 getService 方法做了什么：

```
frameworks/base/core/java/android/app/ActivityManager.java
```

```
public static IActivityManager getService() {
    return IActivityManagerSingleton.get();
}

private static final Singleton<IActivityManager> IActivityManagerSingleton = new
Singleton<IActivityManager>() {
    @Override
    protected IActivityManager create() {
        final IBinder b = ServiceManager.getService(Context.ACTIVITY_
SERVICE);//1
        final IActivityManager am = IActivityManager.Stub.asInterface(b);//2
        return am;
    }
};
```

getService 方法调用了 IActivityManagerSingleton 的 get 方法，我们接着往下看，IActivityManagerSingleton 是一个 Singleton 类。在注释 1 处得到名为“activity”的 Service 引用，也就是 IBinder 类型的 AMS 的引用。接着在注释 2 处将它转换成 IActivityManager 类型的对象，这段代码采用的是 AIDL，IActivityManager.java 类是由 AIDL 工具在编译时自动生成的，IActivityManager.aidl 的文件路径为 frameworks/base/core/java/android/app/IActivityManager.aidl。要实现进程间通信，服务器端也就是 AMS 只需要继承 IActivityManager.Stub 类并实现相应的方法就可以了。注意 Android 8.0 之前并没有采用 AIDL，而是采用了类似 AIDL 的形式，用 AMS 的代理对象 ActivityManagerProxy 来与 AMS 进行进程间通信，Android 8.0 去除了 ActivityManagerNative 的内部类 ActivityManagerProxy，代替它的是 IActivityManager，它是 AMS 在本地的代理。回到 Instrumentation 类的 execStartActivity 方法中，从上面得知 execStartActivity 方法最终调用的是 AMS 的 startActivity 方法。

4.1.2 AMS到ApplicationThread的调用过程

Launcher 请求 AMS 后，代码逻辑已经进入 AMS 中，接着是 AMS 到 ApplicationThread 的调用流程，时序图如图 4-2 所示。

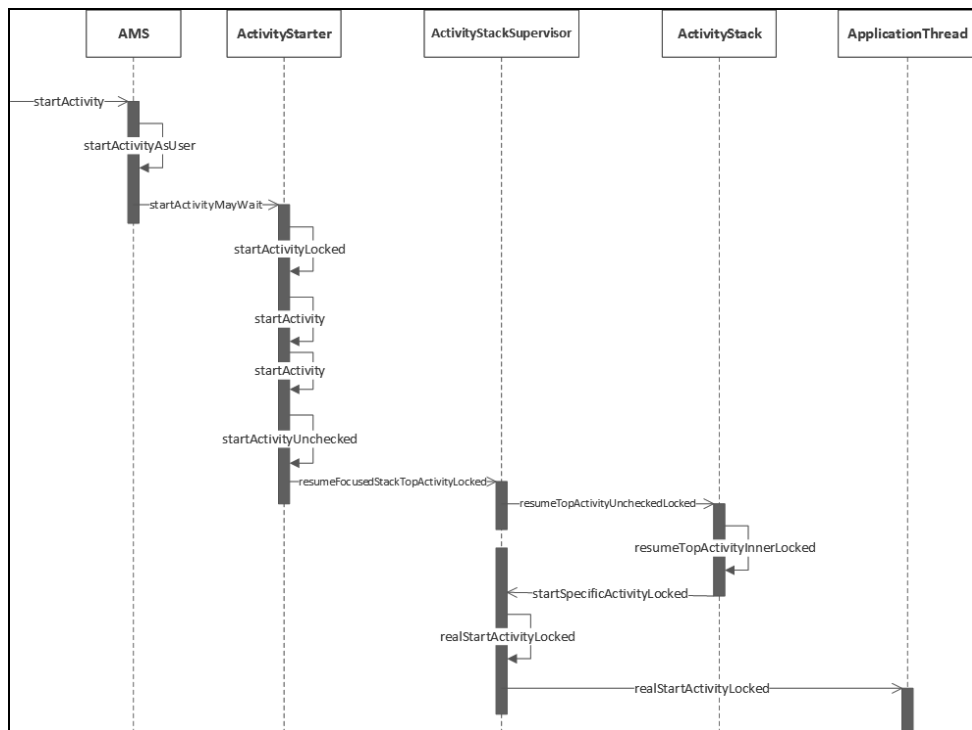


图 4-2 AMS 到 ApplicationThread 的调用过程的时序图

AMS 的 startActivity 方法如下所示：

frameworks/base/services/core/java/com/android/server/am/ActivityManagerService.java

```

@Override
public final int startActivity(IApplicationThread caller, String callingPackage,
    Intent intent, String resolvedType, IBinder resultTo, String resultWho,
    int requestCode, int startFlags, ProfilerInfo profilerInfo, Bundle
    bOptions) {
    return startActivityAsUser(caller, callingPackage, intent, resolvedType,
        resultTo, resultWho, requestCode, startFlags, profilerInfo, bOptions,
        UserHandle.getCallingUserId());
}
  
```

在 AMS 的 startActivity 方法中返回了 startActivityAsUser 方法，可以发现 startActivityAsUser 方法比 startActivity 方法多了一个参数 UserHandle.getCallingUserId()，这个方法会获得调用者的 UserId，AMS 根据这个 UserId 来确定调用者的权限。

frameworks/base/services/core/java/com/android/server/am/ActivityManagerService.java

```

@Override
  
```

```

public final int startActivityAsUser(IApplicationThread caller, String callingPackage,
    Intent intent, String resolvedType, IBinder resultTo, String resultWho,
    int requestCode, int startFlags, ProfilerInfo profilerInfo, Bundle bOptions,
    int userId) {
    //判断调用者进程是否被隔离
    enforceNotIsolatedCaller("startActivity");//1
    //检查调用者权限
    userId = mUserController.handleIncomingUser(Binder.getCallingPid(), Binder.
        getCallingUid(), userId, false, ALLOW_FULL_ONLY, "startActivity", null);//2
    return mActivityStarter.startActivityMayWait(caller, -1, callingPackage,
        intent, resolvedType, null, null, resultTo, resultWho, requestCode,
        startFlags, profilerInfo, null, null, bOptions, false, userId, null, null,
        "startActivityAsUser");
}

```

在注释 1 处判断调用者进程是否被隔离，如果被隔离则抛出 `SecurityException` 异常，在注释 2 处检查调用者是否有权限，如果没有权限也会抛出 `SecurityException` 异常。最后调用了 `ActivityStarter` 的 `startActivityLocked` 方法，`startActivityLocked` 方法的参数要比 `startActivityAsUser` 多几个，需要注意的是倒数第二个参数类型为 `TaskRecord`，代表启动的 `Activity` 所在的栈。最后一个参数 `"startActivityAsUser"` 代表启动的理由。`startActivityLocked` 方法的代码如下所示：

```

frameworks/base/services/core/java/com/android/server/am/ActivityStarter.java

final int startActivityMayWait(IApplicationThread caller, int callingUid,
    String callingPackage, Intent intent, String resolvedType,
    IVoiceInteractionSession voiceSession, IVoiceInteractor voiceInteractor,
    IBinder resultTo, String resultWho, int requestCode, int startFlags,
    ProfilerInfo profilerInfo, WaitResult outResult,
    Configuration globalConfig, Bundle bOptions, boolean ignoreTargetSecurity,
    int userId, IActivityContainer iContainer, TaskRecord inTask,
    String reason) {
    ...
    int res = startActivityLocked(caller, intent, ephemeralIntent, resolvedType,
        aInfo, rInfo, voiceSession, voiceInteractor,
        resultTo, resultWho, requestCode, callingPid,
        callingUid, callingPackage, realCallingPid, realCallingUid, startFlags,
        options, ignoreTargetSecurity, componentSpecified, outRecord,
        container, inTask, reason);
    ...
    return res;
}

```

ActivityStarter 是 Android 7.0 中新加入的类，它是加载 Activity 的控制类，会收集所有的逻辑来决定如何将 Intent 和 Flags 转换为 Activity，并将 Activity 和 Task 以及 Stack 相关联。ActivityStarter 的 startActivityMayWait 方法调用了 startActivityLocked 方法，如下所示：

frameworks/base/services/core/java/com/android/server/am/ActivityStarter.java

```
int startActivityLocked(IApplicationThread caller, Intent intent, Intent
ephemeralIntent, String resolvedType, ActivityInfo aInfo, ResolveInfo rInfo,
    IVoiceInteractionSession voiceSession, IVoiceInteractor voiceInteractor,
    IBinder resultTo, String resultWho, int requestCode, int callingPid, int
callingUid, String callingPackage, int realCallingPid, int realCallingUid,
int startFlags, ActivityOptions options, boolean ignoreTargetSecurity,
boolean component Specified, ActivityRecord[] outActivity,
    ActivityStackSupervisor.ActivityContainer container, TaskRecord inTask,
    String reason) {
    //判断启动的理由不为空
    if (TextUtils.isEmpty(reason)) { //1
        throw new IllegalArgumentException("Need to specify a reason.");
    }
    mLastStartReason = reason;
    mLastStartActivityTimeMs = System.currentTimeMillis();
    mLastStartActivityRecord[0] = null;
    mLastStartActivityResult = startActivity(caller, intent, ephemeralIntent,
        resolvedType, aInfo, rInfo, voiceSession, voiceInteractor, resultTo,
        resultWho, requestCode, callingPid, callingUid, callingPackage,
        realCallingPid, realCallingUid, startFlags,
        options, ignoreTargetSecurity, componentSpecified, mLastStartActivityRecord,
        container, inTask);
    if (outActivity != null) {
        outActivity[0] = mLastStartActivityRecord[0];
    }
    return mLastStartActivityResult;
}
```

在注释 1 处判断启动的理由不为空，如果为空则抛出 IllegalArgumentException 异常。紧接着又调用了 startActivity 方法，如下所示：

frameworks/base/services/core/java/com/android/server/am/ActivityStarter.java

```
private int startActivity(IApplicationThread caller, Intent intent, Intent
ephemeralIntent, String resolvedType, ActivityInfo aInfo, ResolveInfo rInfo,
    IVoiceInteractionSession voiceSession, IVoiceInteractor voiceInteractor,
    IBinder resultTo, String resultWho, int requestCode, int callingPid, int
callingUid, String callingPackage, int realCallingPid, int realCallingUid,
int startFlags, ActivityOptions options, boolean ignoreTargetSecurity,
```

```

        boolean componentSpecified, ActivityRecord[] outActivity,
        ActivityStackSupervisor.ActivityContainer container,
        TaskRecord inTask) {
int err = ActivityManager.START_SUCCESS;
final Bundle verificationBundle
    = options != null ? options.popAppVerificationBundle() : null;
ProcessRecord callerApp = null;
if (caller != null) {//1
    //得到 Launcher 进程
    callerApp = mService.getRecordForAppLocked(caller);//2
    if (callerApp != null) {
        //获取 Launcher 进程的 pid 和 uid
        callingPid = callerApp.pid;
        callingUid = callerApp.info.uid;
    } else {
        Slog.w(TAG, "Unable to find app for caller " + caller
            + " (pid=" + callingPid + ") when starting: "
            + intent.toString());
        err = ActivityManager.START_PERMISSION_DENIED;
    }
}
...
//创建即将要启动的 Activity 的描述类 ActivityRecord
ActivityRecord r = new ActivityRecord(mService, callerApp, callingPid,
callingUid, callingPackage, intent, resolvedType, aInfo, mService.getGlobal
Configuration(),resultRecord, resultWho, requestCode, componentSpecified,
voiceSession != null, mSupervisor, container, options, sourceRecord);
if (outActivity != null) {
    outActivity[0] = r;//3
}
...
doPendingActivityLaunchesLocked(false);
return startActivity(r, sourceRecord, voiceSession, voiceInteractor,
startFlags, true, options, inTask, outActivity);//4
}

```

ActivityStarter 的 startActivity 方法逻辑比较多，这里列出部分我们需要关心的代码。在注释 1 处判断 IApplicationThread 类型的 caller 是否为 null，这个 caller 是方法调用一路传过来的，指向的是 Launcher 所在的应用程序进程的 ApplicationThread 对象，在注释 2 处调用 AMS 的 getRecordForAppLocked 方法得到的是代表 Launcher 进程的 callerApp 对象，它是 ProcessRecord 类型的，ProcessRecord 用于描述一个应用程序进程。同样地，ActivityRecord 用于描述一个 Activity，用来记录一个 Activity 的所有信息。接下来创建 ActivityRecord，

用于描述将要启动的 Activity,并在注释 3 处将创建的 ActivityRecord 赋值给 ActivityRecord[] 类型的 outActivity, 这个 outActivity 会作为注释 4 处的 startActivity 方法的参数传递下去。

```
frameworks/base/services/core/java/com/android/server/am/ActivityStarter.java
```

```
private int startActivity(final ActivityRecord r, ActivityRecord sourceRecord,
    IVoiceInteractionSession voiceSession, IVoiceInteractor voiceInteractor,
        int startFlags, boolean doResume, ActivityOptions options, TaskRecord
        inTask, ActivityRecord[] outActivity) {
    int result = START_CANCELED;
    try {
        mService.mWindowManager.deferSurfaceLayout();
        result = startActivityUnchecked(r, sourceRecord, voiceSession, voiceInteractor,
            startFlags, doResume, options, inTask, outActivity);
    }
    ...
    return result;
}
```

startActivity 方法紧接着调用了 startActivityUnchecked 方法:

```
frameworks/base/services/core/java/com/android/server/am/ActivityStarter.java
```

```
private int startActivityUnchecked(final ActivityRecord r, ActivityRecord
    sourceRecord, IVoiceInteractionSession voiceSession, IVoiceInteractor
    voiceInteractor, int startFlags, boolean doResume, ActivityOptions options,
    TaskRecord inTask, ActivityRecord[] outActivity) {
    ...
    if (mStartActivity.resultTo == null && mInTask == null && !mAddingToTask
        && (mLaunchFlags & FLAG_ACTIVITY_NEW_TASK) != 0) { //1
        newTask = true;
        //创建新的 TaskRecord
        result = setTaskFromReuseOrCreateNewTask(
            taskToAffiliate, preferredLaunchStackId, topStack); //2
    } else if (mSourceRecord != null) {
        result = setTaskFromSourceRecord();
    } else if (mInTask != null) {
        result = setTaskFromInTask();
    } else {
        setTaskToCurrentTopOrCreateNewTask();
    }
    ...
    if (mDoResume) {
        final ActivityRecord topTaskActivity =
            mStartActivity.getTask().topRunningActivityLocked();
        if (!mTargetStack.isFocusable())
```



```

        || (topTaskActivity != null && topTaskActivity.mTaskOverlay
        && mStartActivity != topTaskActivity)) {
        ...
    } else {
        if (mTargetStack.isFocusable() && !mSupervisor.isFocusedStack(mTargetStack)) {
            mTargetStack.moveToFront("startActivityUnchecked");
        }
        mSupervisor.resumeFocusedStackTopActivityLocked(mTargetStack,
        mStartActivity, mOptions); //3
    }
} else {
    mTargetStack.addRecentActivityLocked(mStartActivity);
}
...
}

```

startActivityUnchecked 方法主要处理与栈管理相关的逻辑。在标注①处我们得知，启动根 Activity 时会将 Intent 的 Flag 设置为 FLAG_ACTIVITY_NEW_TASK，这样注释 1 处的条件判断就会满足，接着执行注释 2 处的 setTaskFromReuseOrCreateNewTask 方法，其内部会创建一个新的 TaskRecord，用来描述一个 Activity 任务栈，也就是说 setTaskFromReuseOrCreateNewTask 方法内部会创建一个新的 Activity 任务栈。Activity 任务栈其实是一个假想的模型，并不真实存在，关于 Activity 任务栈会在第 6 章进行介绍。在注释 3 处会调用 ActivityStackSupervisor 的 resumeFocusedStackTopActivityLocked 方法，如下所示：

```

frameworks/base/services/core/java/com/android/server/am/ActivityStackSupervisor.java

boolean resumeFocusedStackTopActivityLocked(
    ActivityStack targetStack, ActivityRecord target, ActivityOptions targetOptions) {
    if (targetStack != null && isFocusedStack(targetStack)) {
        return targetStack.resumeTopActivityUncheckedLocked(target, targetOptions);
    }
    //获取要启动的 Activity 所在栈的栈顶的不是处于停止状态的 ActivityRecord
    final ActivityRecord r = mFocusedStack.topRunningActivityLocked(); //1
    if (r == null || r.state != RESUMED) { //2
        mFocusedStack.resumeTopActivityUncheckedLocked(null, null); //3
    } else if (r.state == RESUMED) {
        mFocusedStack.executeAppTransition(targetOptions);
    }
    return false;
}

```

在注释 1 处调用 ActivityStack 的 topRunningActivityLocked 方法获取要启动的 Activity 所在栈的栈顶的不是处于停止状态的 ActivityRecord。在注释 2 处，如果 ActivityRecord 不为 null，或者要启动的 Activity 的状态不是 RESUMED 状态，就会调用注释 3 处的 ActivityStack 的 resumeTopActivityUncheckedLocked 方法，对于即将启动的 Activity，注释 2 处的条件判断是肯定满足的，我们来查看 ActivityStack 的 resumeTopActivityUncheckedLocked 方法，如下所示：

```
frameworks/base/services/core/java/com/android/server/am/ActivityStack.java
```

```
boolean resumeTopActivityUncheckedLocked(ActivityRecord prev, ActivityOptions
options) {
    if (mStackSupervisor.inResumeTopActivity) {
        return false;
    }
    boolean result = false;
    try {
        mStackSupervisor.inResumeTopActivity = true;
        result = resumeTopActivityInnerLocked(prev, options);//1
    } finally {
        mStackSupervisor.inResumeTopActivity = false;
    }
    mStackSupervisor.checkReadyForSleepLocked();
    return result;
}
```

紧接着查看注释 1 处 ActivityStack 的 resumeTopActivityInnerLocked 方法：

```
frameworks/base/services/core/java/com/android/server/am/ActivityStack.java
```

```
private boolean resumeTopActivityInnerLocked(ActivityRecord prev, ActivityOptions
options) {
    ...
    mStackSupervisor.startSpecificActivityLocked(next, true, true);
}
if (DEBUG_STACK) mStackSupervisor.validateTopActivitiesLocked();
return true;
}
```

resumeTopActivityInnerLocked 方法代码非常多，我们只需要关注调用了 ActivityStackSupervisor 的 startSpecificActivityLocked 方法即可，代码如下所示：

```
frameworks/base/services/core/java/com/android/server/am/ActivityStackSupervisor.java
```

```
void startSpecificActivityLocked(ActivityRecord r,
    boolean andResume, boolean checkConfig) {
```

```

//获取即将启动的 Activity 所在的应用程序进程
ProcessRecord app = mService.getProcessRecordLocked(r.processName,
    r.info.applicationInfo.uid, true);//1
r.getStack().setLaunchTime(r);

if (app != null && app.thread != null) {//2
    try {
        if ((r.info.flags&ActivityInfo.FLAG_MULTIPROCESS) == 0
            || !"android".equals(r.info.packageName)) {
            app.addPackage(r.info.packageName, r.info.applicationInfo.versionCode,
                mService.mProcessStats);
        }
        realStartActivityLocked(r, app, andResume, checkConfig);//3
        return;
    } catch (RemoteException e) {
        Slog.w(TAG, "Exception when starting activity "
            + r.intent.getComponent().flattenToShortString(), e);
    }
}
mService.startProcessLocked(r.processName, r.info.applicationInfo, true, 0,
    "activity", r.intent.getComponent(), false, false, true);
}

```

在注释 1 处获取即将启动的 Activity 所在的应用程序进程，在注释 2 处判断要启动的 Activity 所在的应用程序进程如果已经运行的话，就会调用注释 3 处的 `realStartActivityLocked` 方法，这个方法的第二个参数是代表要启动的 Activity 所在的应用程序进程的 `ProcessRecord`。

```
frameworks/base/services/core/java/com/android/server/am/ActivityStackSupervisor.java
```

```

final boolean realStartActivityLocked(ActivityRecord r, ProcessRecord app,
    boolean andResume, boolean checkConfig) throws RemoteException {
    ...
    app.thread.scheduleLaunchActivity(new Intent(r.intent), r.appToken,
        System.identityHashCode(r), r.info, new Configuration(mService.
            mConfiguration), new Configuration(task.mOverrideConfig), r.compat,
            r.launchFromPackage, task.voiceInteractor, app.repProcState,
            r.icicle, r.persistentState, results, newIntents, !andResume,
            mService.isNextTransitionForward(), profilerInfo);
    ...
    return true;
}

```

这里的 `app.thread` 指的是 `IApplicationThread`，它的实现是 `ActivityThread` 的内部类 `ApplicationThread`，其中 `ApplicationThread` 继承了 `IApplicationThread.Stub`。`app` 指的是传入

的要启动的 Activity 所在的应用程序进程，因此，这段代码指的就是要在目标应用程序进程启动 Activity。当前代码逻辑运行在 AMS 所在的进程（SystemServer 进程）中，通过 ApplicationThread 来与应用程序进程进行 Binder 通信，换句话说，ApplicationThread 是 AMS 所在进程（SystemServer 进程）和应用程序进程的通信桥梁，如图 4-3 所示。

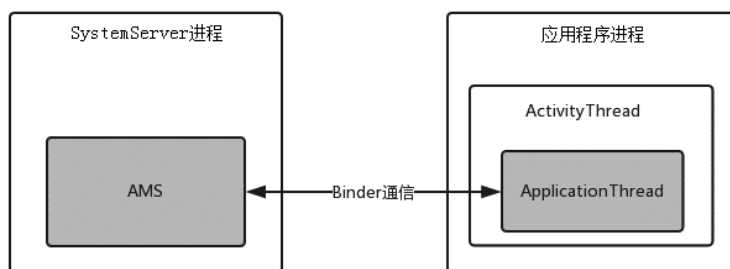


图 4-3 AMS 与应用程序进程通信

4.1.3 ActivityThread启动Activity的过程

通过 4.1.2 节的知识，我们知道目前的代码逻辑运行在应用程序进程中。先来查看 ActivityThread 启动 Activity 过程的时序图，如图 4-4 所示。

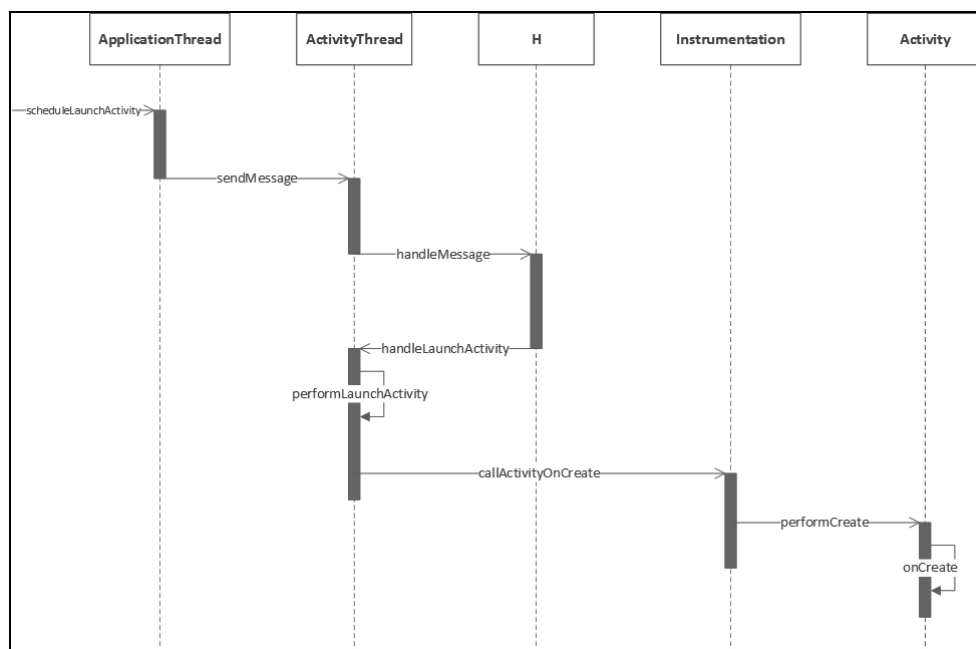


图 4-4 ActivityThread 启动 Activity 过程的时序图

接着查看 `ApplicationThread` 的 `scheduleLaunchActivity` 方法，其中 `ApplicationThread` 是 `ActivityThread` 的内部类，在 3.2.2 节中讲过应用程序进程创建后会运行代表主线程的实例 `ActivityThread`，它管理着当前应用程序进程的主线程。`ApplicationThread` 的 `scheduleLaunchActivity` 方法如下所示：

```
frameworks/base/core/java/android/app/ActivityThread.java
```

```
@Override
public final void scheduleLaunchActivity(Intent intent, IBinder token, int
    ident, ActivityInfo info, Configuration curConfig, Configuration overrideConfig,
    CompatibilityInfo compatInfo, String referrer, IVoiceInteractor
    voiceInteractor, int procState, Bundle state, PersistableBundle
    persistentState, List<ResultInfo> pendingResults, List<ReferrerIntent>
    pendingNewIntents, boolean notResumed, boolean isForward, ProfilerInfo
    profilerInfo) {
    updateProcessState(procState, false);
    ActivityClientRecord r = new ActivityClientRecord();
    r.token = token;
    r.ident = ident;
    r.intent = intent;
    r.referrer = referrer;
    ...
    updatePendingConfiguration(curConfig);
    sendMessage(H.LAUNCH_ACTIVITY, r);
}
```

`scheduleLaunchActivity` 方法将启动 `Activity` 的参数封装成 `ActivityClientRecord`，`sendMessage` 方法向 `H` 类发送类型为 `LAUNCH_ACTIVITY` 的消息，并将 `ActivityClientRecord` 传递过去，`sendMessage` 方法有多个重载方法，最终调用的 `sendMessage` 方法如下所示：

```
frameworks/base/core/java/android/app/ActivityThread.java
```

```
private void sendMessage(int what, Object obj, int arg1, int arg2, boolean async) {
    if (DEBUG_MESSAGES) Slog.v(
        TAG, "SCHEDULE " + what + " " + mH.codeToString(what)
        + ": " + arg1 + " / " + obj);
    Message msg = Message.obtain();
    msg.what = what;
    msg.obj = obj;
    msg.arg1 = arg1;
    msg.arg2 = arg2;
    if (async) {
        msg.setAsynchronous(true);
    }
}
```

```

        mH.sendMessage(msg);
    }

```

这里 mH 指的是 H，它是 ActivityThread 的内部类并继承自 Handler，是应用程序进程中主线程的消息管理类。因为 ApplicationThread 是一个 Binder，它的调用逻辑运行在 Binder 线程池中，所以这里需要用 H 将代码的逻辑切换到主线程中。H 的代码如下所示：

```

private class H extends Handler {
    public static final int LAUNCH_ACTIVITY = 100;
    public static final int PAUSE_ACTIVITY = 101;
    ...
    public void handleMessage(Message msg) {
        if (DEBUG_MESSAGES) Slog.v(TAG, ">>> handling: " + codeToString(msg.what));
        switch (msg.what) {
            case LAUNCH_ACTIVITY: {
                Trace.traceBegin(Trace.TRACE_TAG_ACTIVITY_MANAGER, "activityStart");
                final ActivityClientRecord r = (ActivityClientRecord) msg.obj; //1
                r.packageInfo = getPackageInfoNoCheck(
                        r.activityInfo.applicationInfo, r.compatInfo); //2
                handleLaunchActivity(r, null, "LAUNCH_ACTIVITY"); //3
                Trace.traceEnd(Trace.TRACE_TAG_ACTIVITY_MANAGER);
            } break;
            case RELAUNCH_ACTIVITY: {
                Trace.traceBegin(Trace.TRACE_TAG_ACTIVITY_MANAGER, "activityRestart");
                ActivityClientRecord r = (ActivityClientRecord) msg.obj;
                handleRelaunchActivity(r);
                Trace.traceEnd(Trace.TRACE_TAG_ACTIVITY_MANAGER);
            } break;
            ...
        }
    }
    ...
}

```

查看 H 的 handleMessage 方法中对 LAUNCH_ACTIVITY 的处理，在注释 1 处将传过来的 msg 的成员变量 obj 转换为 ActivityClientRecord。在注释 2 处通过 getPackageInfoNoCheck 方法获得 LoadedApk 类型的对象并赋值给 ActivityClientRecord 的成员变量 packageInfo。应用程序进程要启动 Activity 时需要将该 Activity 所属的 APK 加载进来，而 LoadedApk 就是用来描述已加载的 APK 文件的。在注释 3 处调用 handleLaunchActivity 方法，代码如下所示：

```
frameworks/base/core/java/android/app/ActivityThread.java
```

```

private void handleLaunchActivity(ActivityClientRecord r, Intent customIntent,
    String reason) {

```

```

...
WindowManagerGlobal.initialize();
//启动 Activity
Activity a = performLaunchActivity(r, customIntent); //1
if (a != null) {
    r.createdConfig = new Configuration(mConfiguration);
    reportSizeConfigurations(r);
    Bundle oldState = r.state;
    //将 Activity 的状态置为 Resume
    handleResumeActivity(r.token, false, r.isForward,
        !r.activity.mFinished && !r.startsNotResumed, r.lastProcessedSeq,
        reason); //2
    if (!r.activity.mFinished && r.startsNotResumed) {
        performPauseActivityIfNeeded(r, reason);
        if (r.isPreHoneycomb()) {
            r.state = oldState;
        }
    }
} else {
    try {
        //停止 Activity 启动
        ActivityManager.getService()
            .finishActivity(r.token, Activity.RESULT_CANCELED, null,
                Activity.DONT_FINISH_TASK_WITH_ACTIVITY);
    } catch (RemoteException ex) {
        throw ex.rethrowFromSystemServer();
    }
}
}

```

注释 1 处的 `performLaunchActivity` 方法用来启动 Activity，注释 2 处的代码用来将 Activity 的状态设置为 Resume。如果该 Activity 为 null 则会通知 AMS 停止启动 Activity。下面来查看 `performLaunchActivity` 方法做了什么：

frameworks/base/core/java/android/app/ActivityThread.java

```

private Activity performLaunchActivity(ActivityClientRecord r, Intent customIntent) {
    //获取 ActivityInfo 类
    ActivityInfo aInfo = r.activityInfo; //1
    if (r.packageInfo == null) {
        //获取 APK 文件的描述类 LoadedApk
        r.packageInfo = getPackageInfo(aInfo.applicationInfo, r.compatInfo,
            Context.CONTEXT_INCLUDE_CODE); //2
    }
}

```

```

ComponentName component = r.intent.getComponent();//3
...
//创建要启动 Activity 的上下文环境
ContextImpl appContext = createBaseContextForActivity(r);//4
Activity activity = null;
try {
    java.lang.ClassLoader cl = appContext.getClassLoader();
    //用类加载器来创建该 Activity 的实例
    activity = mInstrumentation.newActivity(
        cl, component.getClassName(), r.intent);//5
    ...
} catch (Exception e) {
    ...
}
try {
    //创建 Application
    Application app = r.packageInfo.makeApplication(false, mInstrumentation);//6
    ...
    if (activity != null) {
        ...
        /**
         *7 初始化 Activity
         */
        activity.attach(appContext, this, getInstrumentation(), r.token,
            r.ident, app, r.intent, r.activityInfo, title, r.parent, r.embeddedID,
            r.lastNonConfigurationInstances, config, r.referrer, r.voiceInteractor,
            window, r.configCallback );
        ...
        if (r.isPersistable()) {
            mInstrumentation.callActivityOnCreate(activity, r.state,
                r.persistentState);//8
        } else {
            mInstrumentation.callActivityOnCreate(activity, r.state);
        }
        ...
    }
    r.paused = true;
    mActivities.put(r.token, r);
} catch (SuperNotCalledException e) {
    throw e;
} catch (Exception e) {
    ...
}

```



```

        return activity;
    }

```

注释 1 处用来获取 ActivityInfo，用于存储代码以及 AndroidManifest 设置的 Activity 和 Receiver 节点信息，比如 Activity 的 theme 和 launchMode。在注释 2 处获取 APK 文件的描述类 LoadedApk。在注释 3 处获取要启动的 Activity 的 ComponentName 类，在 ComponentName 类中保存了该 Activity 的包名和类名。注释 4 处用来创建要启动 Activity 的上下文环境。注释 5 处根据 ComponentName 中存储的 Activity 类名，用类加载器来创建该 Activity 的实例。注释 6 处用来创建 Application，makeApplication 方法内部会调用 Application 的 onCreate 方法。注释 7 处调用 Activity 的 attach 方法初始化 Activity，在 attach 方法中会创建 Window 对象（PhoneWindow）并与 Activity 自身进行关联。在注释 8 处调用 Instrumentation 的 callActivityOnCreate 方法来启动 Activity，如下所示：

```
frameworks/base/core/java/android/app/Instrumentation.java
```

```

public void callActivityOnCreate(Activity activity, Bundle icle,
    PersistableBundle persistentState) {
    prePerformCreate(activity);
    activity.performCreate(icle, persistentState); //1
    postPerformCreate(activity);
}

```

注释 1 处调用了 Activity 的 performCreate 方法，代码如下所示：

```
frameworks/base/core/java/android/app/Activity.java
```

```

final void performCreate(Bundle icle, PersistableBundle persistentState) {
    restoreHasCurrentPermissionRequest(icle);
    onCreate(icle, persistentState);
    mActivityTransitionState.readState(icle);
    performCreateCommon();
}

```

在 performCreate 方法中会调用 Activity 的 onCreate 方法，讲到这里，根 Activity 就启动了，即应用程序就启动了。根 Activity 启动过程就讲到这里，下面我们来学习根 Activity 启动过程中涉及的进程。

4.1.4 根Activity启动过程中涉及的进程

根 Activity 启动过程中会涉及 4 个进程，分别是 Zygote 进程、Launcher 进程、AMS 所在进程（SystemServer 进程）、应用程序进程。它们之间的关系如图 4-5 所示。

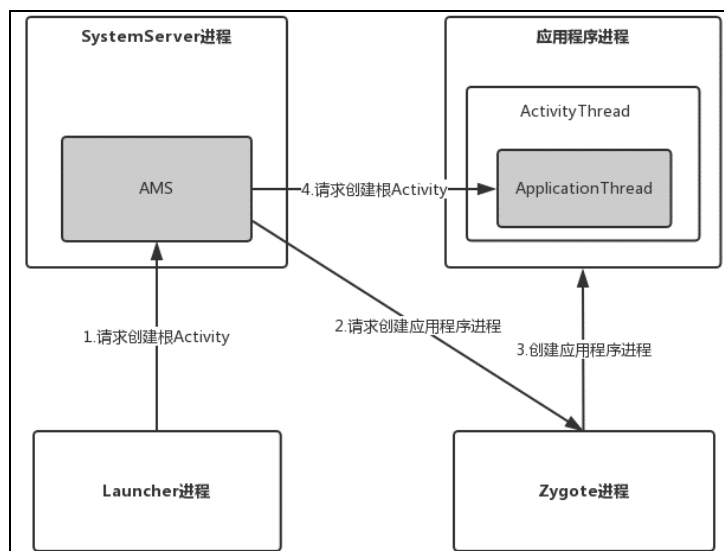


图 4-5 根 Activity 启动过程中涉及的进程之间的关系

图 4-5 在图 4-3 基础上进行了修改，首先 Launcher 进程向 AMS 请求创建根 Activity，AMS 会判断根 Activity 所需的应用程序进程是否存在并启动，如果不存在就会请求 Zygote 进程创建应用程序进程。应用程序进程启动后，AMS 会请求创建应用程序进程并启动根 Activity。图 4-5 中步骤 2 采用的是 Socket 通信，步骤 1 和步骤 4 采用的是 Binder 通信。图 4-5 可能并不是很直观，为了更好理解，下面给出这 4 个进程调用的时序图，如图 4-6 所示。

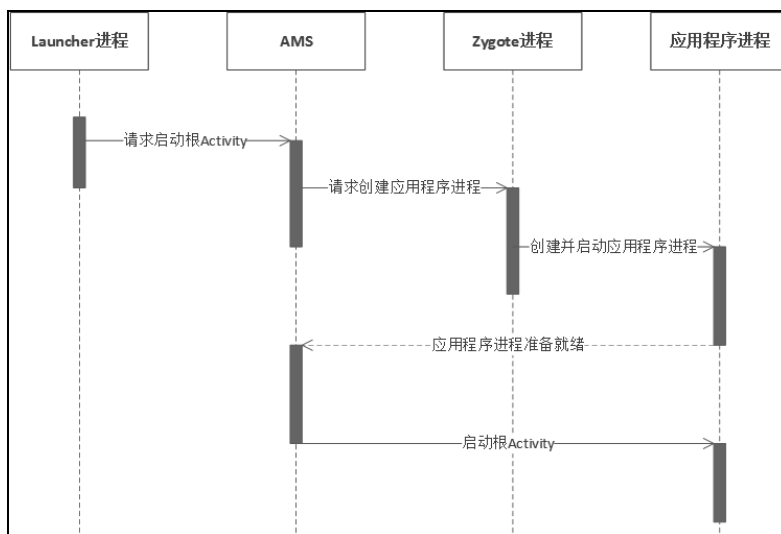


图 4-6 根 Activity 启动过程中进程调用时序图

如果是普通 Activity 启动过程会涉及几个进程呢？答案是两个，AMS 所在进程和应用程序进程。实际上理解了根 Activity 的启动过程（根 Activity 的 onCreate 过程），根 Activity 和普通 Activity 其他生命周期状态（比如 onStart、onResume 等）过程也会很轻松掌握，这些知识点都是触类旁通的，想要具体了解这些知识点的读者可以自行阅读源码，由于篇幅有限这里就不再介绍了。

4.2 Service的启动过程

Service 的启动过程和根 Activity 启动过程有部分相似的知识点，另外 Service 启动过程涉及上下文 Context 的知识点，这里只关注流程而不会详细介绍 Context，关于上下文 Context 会在第 5 章进行介绍。Service 的启动过程将分为两个部分来进行讲解，分别是 ContextImpl 到 ActivityManageService 的调用过程和 ActivityThread 启动 Service。

4.2.1 ContextImpl到AMS的调用过程

ContextImpl 到 AMS 的调用过程很简短，如图 4-7 所示。

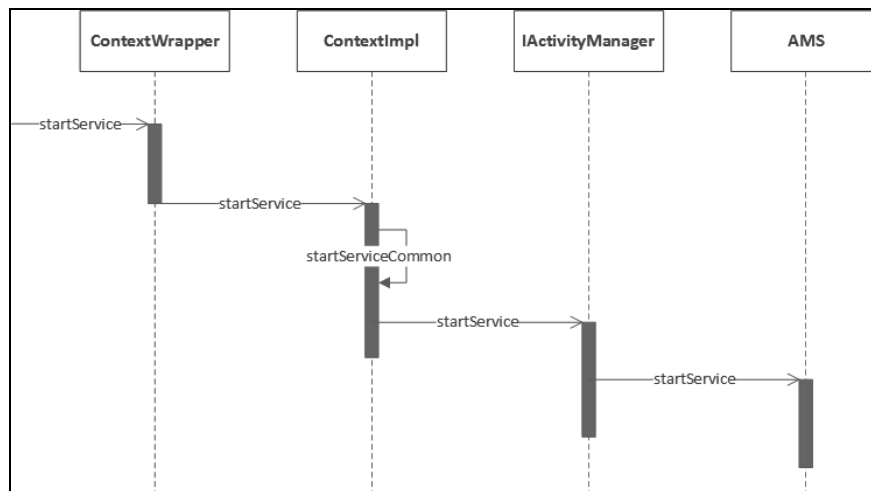


图 4-7 ContextImpl 到 AMS 的调用过程

要启动 Service，我们会调用 startService 方法，它在 ContextWrapper 中实现，代码如下所示：

```
frameworks/base/core/java/android/content/ContextWrapper.java
```

```
public class ContextWrapper extends Context {
    Context mBase;
    ...
    @Override
    public ComponentName startService(Intent service) {
        return mBase.startService(service);
    }
    ...
}
```

在 startService 方法中会调用 mBase 的 startService 方法，Context 类型的 mBase 对象具体指的是什么呢？在 4.1.3 节中我们讲过，ActivityThread 启动 Activity 时会调用如下代码创建 Activity 的上下文环境：

```
frameworks/base/core/java/android/app/ActivityThread.java
```

```
private Activity performLaunchActivity(ActivityClientRecord r, Intent customIntent) {
    ...
    ContextImpl appContext = createBaseContextForActivity(r);//1
    if (activity != null) {
        ...
        activity.attach(appContext, this, getInstrumentation(), r.token,
            r.ident, app, r.intent, r.activityInfo, title, r.parent,
            r.embeddedID, r.lastNonConfigurationInstances, config,
            r.referrer, r.voiceInteractor, window);
        ...
    }
    ...
    return activity;
}
```

在注释 1 处创建上下文对象 appContext，并传入 Activity 的 attach 方法中，将 Activity 与上下文对象 appContext 关联起来，这个上下文对象 appContext 的具体类型是什么？我们接着查看 createBaseContextForActivity 方法，代码如下所示：

```
frameworks/base/core/java/android/app/ActivityThread.java
```

```
private ContextImpl createBaseContextForActivity(ActivityClientRecord r) {
    ...
    ContextImpl appContext = ContextImpl.createActivityContext(
        this, r.packageInfo, r.activityInfo, r.token, displayId, r.overrideConfig);
    ...
    return appContext;
}
```

上下文对象 `appContext` 的具体类型就是 `ContextImpl`，在 `Activity` 的 `attach` 方法中将 `ContextImpl` 赋值给 `ContextWrapper` 的成员变量 `mBase`，因此，上面提出的问题就得到了解答，`mBase` 具体指向的就是 `ContextImpl`。那么，紧接着来查看 `ContextImpl` 的 `startService` 方法，代码如下所示：

```
frameworks/base/core/java/android/app/ContextImpl.java
```

```
@Override
public ComponentName startService(Intent service) {
    warnIfCallingFromSystemProcess();
    return startServiceCommon(service, mUser);
}

private ComponentName startServiceCommon(Intent service, boolean requireForeground,
    UserHandle user) {
    try {
        validateServiceIntent(service);
        service.prepareToLeaveProcess(this);
        /**
         * 1
         */
        ComponentName cn = ActivityManager.getService().startService(
            mMainThread.getApplicationThread(), service,
            service.resolveTypeIfNeeded (getContentResolver()),
            requireForeground, getOpPackageName(), user.getIdentifier());
        ...
        return cn;
    } catch (RemoteException e) {
        throw e.rethrowFromSystemServer();
    }
}
```

在 `startService` 方法中会返回 `startServiceCommon` 方法，在 `startServiceCommon` 方法中会在注释 1 处调用 AMS 的代理 `IActivityManager` 的 `startService` 方法，最终调用的是 AMS 的 `startService` 方法，这一过程已经在 4.1.1 节讲过了，这里不再赘述。

4.2.2 ActivityThread启动Service

`ActivityThread` 启动 `Service` 的时序图如图 4-8 所示。

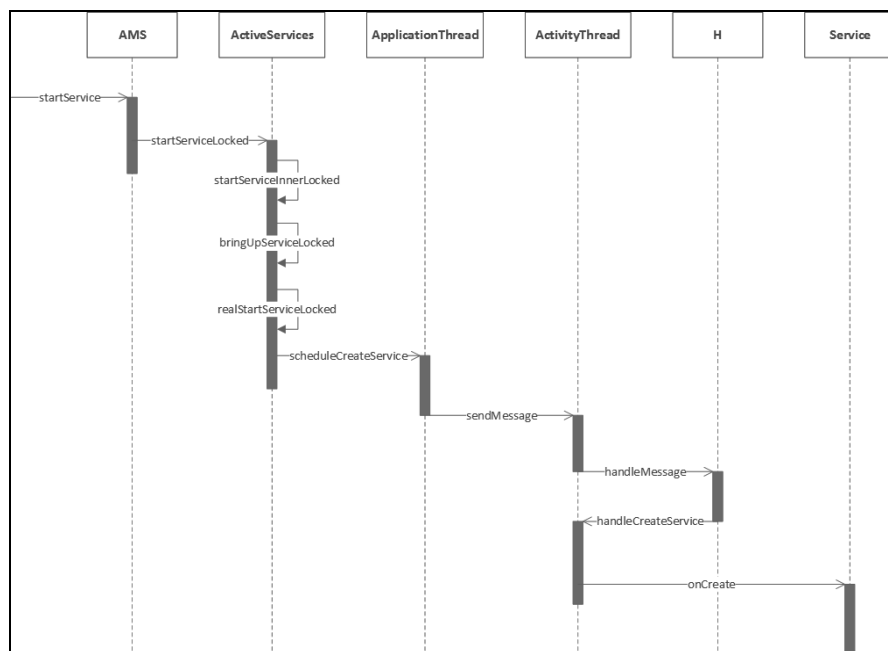


图 4-8 ActivityThread 启动 Service 的时序图

接着我们来查看 AMS 的 startService 方法，如下所示：

frameworks/base/services/core/java/com/android/server/am/ActivityManagerService.java

```

@Override
public ComponentName startService(IApplicationThread caller, Intent service,
    String resolvedType, boolean requireForeground, String callingPackage, int
    userId) throws TransactionTooLargeException {
    ...
    synchronized(this) {
        final int callingPid = Binder.getCallingPid();
        final int callingUid = Binder.getCallingUid();
        final long origId = Binder.clearCallingIdentity();
        ComponentName res;
        try {
            res = mServices.startServiceLocked(caller, service,
                resolvedType, callingPid, callingUid,
                requireForeground, callingPackage, userId);//1
        } finally {
            Binder.restoreCallingIdentity(origId);
        }
        return res;
    }
}

```

注释1处调用 mServices 的 startServiceLocked 方法, mServices 的类型是 ActiveServices, ActiveServices 的 startServiceLocked 方法代码如下所示:

```
frameworks/base/services/core/java/com/android/server/am/ActiveServices.java
```

```
ComponentName startServiceLocked(IApplicationThread caller, Intent service,
String resolvedType,
    int callingPid, int callingUid, boolean fgRequired, String callingPackage,
    final int userId)
    throws TransactionTooLargeException {
    ...
    ServiceLookupResult res =
        retrieveServiceLocked(service, resolvedType, callingPackage,
            callingPid, callingUid, userId, true, callerFg, false); //1
    if (res == null) {
        return null;
    }
    if (res.record == null) {
        return new ComponentName("!", res.permission != null
            ? res.permission : "private to package");
    }
    ServiceRecord r = res.record; //2
    ...
    ComponentName cmp = startServiceInnerLocked(smap, service, r, callerFg, addToStarting);
    return cmp;
}
```

注释1处的 retrieveServiceLocked 方法会查找是否有与参数 service 对应的 ServiceRecord, 如果没有找到, 就会调用 PackageManagerService 去获取参数 service 对应的 Service 信息, 并封装到 ServiceRecord 中, 最后将 ServiceRecord 封装为 ServiceLookupResult 返回。其中 ServiceRecord 用于描述一个 Service, 和此前讲过的 ActivityRecord 类似。在注释2处通过注释1处返回的 ServiceLookupResult 得到参数 service 对应的 ServiceRecord, 并传入到注释3处的 startServiceInnerLocked 方法中。

```
frameworks/base/services/core/java/com/android/server/am/ActiveServices.java
```

```
ComponentName startServiceInnerLocked(ServiceMap smap, Intent service, ServiceRecord r,
    boolean callerFg, boolean addToStarting) throws TransactionTooLarge
    Exception {
    ...
    String error = bringUpServiceLocked(r, service.getFlags(), callerFg, false,
        false);
    if (error != null) {
        return new ComponentName("!!", error);
    }
}
```

```

    }
    ...
    return r.name;
}

```

在 `startServiceInnerLocked` 方法中又调用了 `bringUpServiceLocked` 方法，如下所示：

frameworks/base/services/core/java/com/android/server/am/ActiveServices.java

```

private String bringUpServiceLocked(ServiceRecord r, int intentFlags, boolean
execInFg, boolean whileRestarting, boolean permissionsReviewRequired)
throws TransactionTooLargeException {
    ...
    //获取 Service 想要在哪进程中运行
    final String procName = r.processName; //1
    String hostingType = "service";
    ProcessRecord app;
    if (!isolated) {
        app = mAm.getProcessRecordLocked(procName, r.appInfo.uid, false); //2
        if (DEBUG_MU) Slog.v(TAG_MU, "bringUpServiceLocked: appInfo.uid=" +
            r.appInfo.uid + " app=" + app);
        //如果运行 Service 的应用程序进程存在
        if (app != null && app.thread != null) { //3
            try {
                app.addPackage(r.appInfo.packageName, r.appInfo.versionCode, mAm.
                    mProcessStats);
                //启动 Service
                realStartServiceLocked(r, app, execInFg); //4
                return null;
            } catch (TransactionTooLargeException e) {
                throw e;
            } catch (RemoteException e) {
                Slog.w(TAG, "Exception when starting service " + r.shortName, e);
            }
        }
    } else {
        app = r.isolatedProc;
        if (WebViewZygote.isMultiprocessEnabled()
            && r.serviceInfo.packageName.equals(WebViewZygote.getPackageName())) {
            hostingType = "webview_service";
        }
    }
    //如果用来运行 Service 的应用程序进程不存在
    if (app == null && !permissionsReviewRequired) { //5
        //创建应用程序进程

```



```

        if ((app=mAm.startProcessLocked(procName, r.appInfo, true, intentFlags,
        hostingType, r.name, false, isolated, false)) == null) { //6
            String msg = "Unable to launch app "
                + r.appInfo.packageName + "/"
                + r.appInfo.uid + " for service "
                + r.intent.getIntent() + ": process is bad";
            Slog.w(TAG, msg);
            bringDownServiceLocked(r);
            return msg;
        }
        if (isolated) {
            r.isolatedProc = app;
        }
    }
    ...
    return null;
}

```

在注释 1 处得到 ServiceRecord 的 processName 值并赋给 procName，其中 processName 用来描述 Service 想要在哪个进程中运行，默认是当前进程，我们也可以在 AndroidManifest 文件中设置 android:process 属性来新开启一个进程运行 Service。在注释 2 处将 procName 和 Service 的 uid 传入到 AMS 的 getProcessRecordLocked 方法中，查询是否存在一个与 Service 对应的 ProcessRecord 类型的对象 app，ProcessRecord 主要用来描述运行的应用程序进程的信息。在注释 5 处判断 Service 对应的 app 为 null 则说明用来运行 Service 的应用程序进程不存在，则调用注释 6 处的 AMS 的 startProcessLocked 方法来创建对应的应用程序进程，关于创建应用程序进程请查看第 3 章的内容，这里只讨论没有设置 android:process 属性，即应用程序进程存在的情况。在注释 3 处判断如果用来运行 Service 的应用程序进程存在，则调用注释 4 处的 realStartServiceLocked 方法来启动 Service：

frameworks/base/services/core/java/com/android/server/am/ActiveServices.java

```

private final void realStartServiceLocked(ServiceRecord r,
    ProcessRecord app, boolean execInFg) throws RemoteException {
    ...
    try {
        ...
        app.thread.scheduleCreateService(r, r.serviceInfo,
            mAm.compatibilityInfoForPackageLocked(r.serviceInfo.applicationInfo),
            app.repProcState);
        r.postNotification();
        created = true;
    } catch (DeadObjectException e) {

```

```

        Slog.w(TAG, "Application dead when creating service " + r);
        mAm.appDiedLocked(app);
        throw e;
    } finally {
        ...
    }
    ...
}

```

在 `realStartServiceLocked` 方法中调用了 `app.thread` 的 `scheduleCreateService` 方法。其中 `app.thread` 是 `IApplicationThread` 类型的，它的实现是 `ActivityThread` 的内部类 `ApplicationThread`。`ApplicationThread` 的 `scheduleCreateService` 方法如下所示：

frameworks/base/core/java/android/app/ActivityThread.java

```

public final void scheduleCreateService(IBinder token,
    ServiceInfo info, CompatibilityInfo compatInfo, int processState) {
    updateProcessState(processState, false);
    CreateServiceData s = new CreateServiceData();
    s.token = token;
    s.info = info;
    s.compatInfo = compatInfo;
    sendMessage(H.CREATE_SERVICE, s);
}

```

`scheduleLaunchActivity` 方法将启动 `Service` 的参数封装成 `ActivityClientRecord`，`sendMessage` 方法向 `H` 类发送类型为 `CREATE_SERVICE` 的消息，并将 `ActivityClientRecord` 传递过去，这个过程和 4.1.3 节 `ActivityThread` 启动 `Activity` 的过程是类似的。`sendMessage` 方法有多个重载方法，最终调用的 `sendMessage` 方法如下所示：

frameworks/base/core/java/android/app/ActivityThread.java

```

private void sendMessage(int what, Object obj, int arg1, int arg2, boolean async) {
    if (DEBUG_MESSAGES) Slog.v(
        TAG, "SCHEDULE " + what + " " + mH.codeToString(what)
        + ": " + arg1 + " / " + obj);
    Message msg = Message.obtain();
    msg.what = what;
    msg.obj = obj;
    msg.arg1 = arg1;
    msg.arg2 = arg2;
    if (async) {
        msg.setAsynchronous(true);
    }
    mH.sendMessage(msg);
}

```

这里 mH 指的是 H，它是 ActivityThread 的内部类并继承自 Handler，是应用程序进程中主线程的消息管理类。我们接着查看 H 的 handleMessage 方法：

```
frameworks/base/core/java/android/app/ActivityThread.java
```

```
private class H extends Handler {
    public static final int LAUNCH_ACTIVITY = 100;
    public static final int PAUSE_ACTIVITY = 101;
    ...
    public void handleMessage(Message msg) {
        if (DEBUG_MESSAGES) Slog.v(TAG, ">>> handling: " + codeToString(msg.what));
        switch (msg.what) {
            ...
            case CREATE_SERVICE:
                Trace.traceBegin(Trace.TRACE_TAG_ACTIVITY_MANAGER,
                    ("serviceCreate: " + String.valueOf(msg.obj)));
                handleCreateService((CreateServiceData)msg.obj);
                Trace.traceEnd(Trace.TRACE_TAG_ACTIVITY_MANAGER);
                break;
            ...
        }
        ...
    }
}
```

handleMessage 方法根据消息类型为 CREATE_SERVICE，会调用 handleCreateService 方法：

```
frameworks/base/core/java/android/app/ActivityThread.java
```

```
private void handleCreateService(CreateServiceData data) {
    unscheduleGcIdler();
    //获取要启动 Service 的应用程序的 LoadedApk
    LoadedApk packageInfo = getPackageInfoNoCheck(
        data.info.applicationInfo, data.compatInfo);//1
    Service service = null;
    try {
        //获取类加载器
        java.lang.ClassLoader cl = packageInfo.getClassLoader();//2
        //创建 Service 实例
        service = (Service) cl.loadClass(data.info.name).newInstance();//3
    } catch (Exception e) {
        ...
    }
}
```

```

try {
    if (localLOGV) Slog.v(TAG, "Creating service " + data.info.name);
    //创建 Service 的上下文环境 ContextImpl 对象
    ContextImpl context = ContextImpl.createAppContext(this, packageInfo);//4
    context.setOuterContext(service);
    Application app = packageInfo.makeApplication(false, mInstrumentation);
    //初始化 Service
    service.attach(context, this, data.info.name, data.token, app,
        ActivityManager.getService());//5
    service.onCreate();//6
    mServices.put(data.token, service);//7
    try {
        ActivityManager.getService().serviceDoneExecuting(
            data.token, SERVICE_DONE_EXECUTING_ANON, 0, 0);
    } catch (RemoteException e) {
        throw e.rethrowFromSystemServer();
    }
} catch (Exception e) {
    ...
}
}

```

在注释 1 处获取要启动 Service 的应用程序的 LoadedApk，LoadedApk 是一个 APK 文件的描述类。在注释 2 处通过调用 LoadedApk 的 getClassLoader 方法来获取类加载器。接着在注释 3 处根据 CreateServiceData 对象中存储的 Service 信息，创建 Service 实例。在注释 4 处创建 Service 的上下文环境 ContextImpl 对象。在注释 5 处通过 Service 的 attach 方法来初始化 Service。在注释 6 处调用 Service 的 onCreate 方法，这样 Service 就启动了。在注释 7 处将启动的 Service 加入到 ActivityThread 的成员变量 mServices 中，其中 mServices 是 ArrayMap 类型。Service 的启动过程就讲到这里，接下来我们学习 Service 的绑定过程。

4.3 Service 的绑定过程

我们可以通过调用 Context 的 startService 来启动 Service，也可以通过 Context 的 bindService 来绑定 Service，绑定 Service 的过程要比启动 Service 的过程复杂一些，建议阅读本节前先阅读上一节 Service 的启动过程，结合着 Service 的启动过程会有更好的理解。关于如何绑定 Service 这种基础的问题，这里并不会讲解。Service 的绑定过程将分为两个部分来进行讲解，分别是 ContextImpl 到 AMS 的调用过程和 Service 的绑定过程。

4.3.1 ContextImpl到AMS的调用过程

ContextImpl 到 AMS 的调用过程如图 4-9 所示。

我们可以用 bindService 方法来绑定 Service，它在 ContextWrapper 中实现，代码如下所示：

```
frameworks/base/core/java/android/content/ContextWrapper.java
```

```
@Override
public boolean bindService(Intent service, ServiceConnection conn,
    int flags) {
    return mBase.bindService(service, conn, flags);
}
```

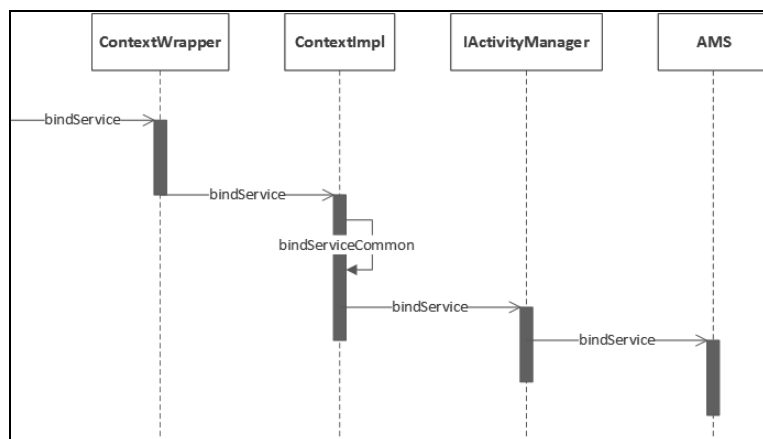


图 4-9 ContextImpl 到 AMS 的调用过程

在 4.2.1 节我们得知 mBase 具体就是指向 ContextImpl 的，接着查看 ContextImpl 的 bindService 方法：

```
frameworks/base/core/java/android/app/ContextImpl.java
```

```
@Override
public boolean bindService(Intent service, ServiceConnection conn,
    int flags) {
    warnIfCallingFromSystemProcess();
    return bindServiceCommon(service, conn, flags, mMainThread.getHandler(),
        Process.myUserHandle());
}
```

在 bindService 方法中，又返回了 bindServiceCommon 方法，代码如下所示：

```
frameworks/base/core/java/android/app/ContextImpl.java
```

```
private boolean bindServiceCommon(Intent service, ServiceConnection conn, int flags,
Handler handler, UserHandle user) {
    IServiceConnection sd;
    if (conn == null) {
        throw new IllegalArgumentException("connection is null");
    }
    if (mPackageInfo != null) {
        sd = mPackageInfo.getServiceDispatcher(conn, getOuterContext(), handler,
flags);//1
    } else {
        throw new RuntimeException("Not supported in system context");
    }
    validateServiceIntent(service);
    try {
        ...
        /**
        * 2
        */
        int res = ActivityManager.getService().bindService(mMainThread.
getApplicationThread(), getActivityToken(), service, service.
resolveTypeIfNeeded(getContentResolver()), sd, flags, getOpPackageName(),
user.getIdentifier());
        if (res < 0) {
            throw new SecurityException(
                "Not allowed to bind to service " + service);
        }
        return res != 0;
    } catch (RemoteException e) {
        throw e.rethrowFromSystemServer();
    }
}
```

在注释 1 处调用了 LoadedApk 类型的对象 mPackageInfo 的 getServiceDispatcher 方法，它的主要作用是将 ServiceConnection 封装为 IServiceConnection 类型的对象 sd，从 IServiceConnection 的名字我们就能得知它实现了 Binder 机制，这样 Service 的绑定就支持了跨进程。接着在注释 2 处我们又看见了熟悉的代码，最终会调用 AMS 的 bindService 方法。

4.3.2 Service 的绑定过程

AMS 的 bindService 方法代码如下所示：

frameworks/base/services/core/java/com/android/server/am/ActivityManagerService.java

```
public int bindService(IApplicationThread caller, IBinder token, Intent service,
    String resolvedType, IServiceConnection connection, int flags, String
    callingPackage, int userId) throws TransactionTooLargeException {
    enforceNotIsolatedCaller("bindService");
    if (service != null && service.hasFileDescriptors() == true) {
        throw new IllegalArgumentException("File descriptors passed in Intent");
    }
    if (callingPackage == null) {
        throw new IllegalArgumentException("callingPackage cannot be null");
    }
    synchronized(this) {
        return mServices.bindServiceLocked(caller, token, service,
            resolvedType, connection, flags, callingPackage, userId);
    }
}
```

bindService 方法最后会调用 ActiveServices 类型的对象 mServices 的 bindServiceLocked 方法:

frameworks/base/services/core/java/com/android/server/am/ActiveServices.java

```
int bindServiceLocked(IApplicationThread caller, IBinder token, Intent service,
    String resolvedType, final IServiceConnection connection, int flags,
    String callingPackage, final int userId) throws TransactionTooLargeException {

    ...
    try {
        ...
        AppBindRecord b = s.retrieveAppBindingLocked(service, callerApp); //1
        ...
        if ((flags & Context.BIND_AUTO_CREATE) != 0) {
            s.lastActivity = SystemClock.uptimeMillis();
            //启动 Service
            if (bringUpServiceLocked(s, service.getFlags(), callerFg, false,
                permissionsReviewRequired) != null) { //2
                return 0;
            }
        }
    }
    ...
    if (s.app != null && b.intent.received) { //3
        try {
            c.conn.connected(s.name, b.intent.binder, false); //4
        } catch (Exception e) {
            Slog.w(TAG, "Failure sending service " + s.shortName
```

```

        + " to connection " + c.conn.asBinder()
        + " (in " + c.binding.client.processName + ")", e);
    }
    if (b.intent.apps.size() == 1 && b.intent.doRebind) { //5
        requestServiceBindingLocked(s, b.intent, callerFg, true); //6
    }
    } else if (!b.intent.requested) { //7
        requestServiceBindingLocked(s, b.intent, callerFg, false); //8
    }
    getServiceMapLocked(s.userId).ensureNotStartingBackgroundLocked(s);
} finally {
    Binder.restoreCallingIdentity(origId);
}
return 1;
}

```

讲到这里,有必要先介绍几个与 Service 相关的对象类型,这样有助于对源码进行理解,如下所示。

- ServiceRecord: 用于描述一个 Service。
- ProcessRecord: 一个进程的信息。
- ConnectionRecord: 用于描述应用程序进程和 Service 建立的一次通信。
- AppBindRecord: 应用程序进程通过 Intent 绑定 Service 时,会通过 AppBindRecord 来维护 Service 与应用程序进程之间的关联。其内部存储了谁绑定的 Service (ProcessRecord)、被绑定的 Service (AppBindRecord)、绑定 Service 的 Intent (IntentBindRecord) 和所有绑定通信记录的信息 (ArraySet<ConnectionRecord>)。
- IntentBindRecord: 用于描述绑定 Service 的 Intent。

在注释 1 处调用了 ServiceRecord 的 retrieveAppBindingLocked 方法来获得 AppBindRecord, retrieveAppBindingLocked 方法内部创建 IntentBindRecord, 并对 IntentBindRecord 的成员变量进行赋值,后面我们会详细介绍这个关键的方法。

在注释 2 处调用 bringUpServiceLocked 方法,在 bringUpServiceLocked 方法中又调用 realStartServiceLocked 方法,最终由 ActivityThread 来调用 Service 的 onCreate 方法启动 Service,这也说明了 bindService 方法内部会启动 Service,启动 Service 这一过程在 4.2.2 节中已经讲过,这里不再赘述。在注释 3 处 s.app != null 表示 Service 已经运行,其中 s 是 ServiceRecord 类型对象, app 是 ProcessRecord 类型对象。b.intent.received 表示当前应用程序进程已经接收到绑定 Service 时返回的 Binder,这样应用程序进程就可以通过 Binder 来获取要绑定的 Service 的访问接口。在注释 4 处调用 c.conn 的 connected 方法,其中 c.conn

指的是 `IServiceConnection`，它的具体实现为 `ServiceDispatcher.InnerConnection`，其中 `ServiceDispatcher` 是 `LoadedApk` 的内部类，`InnerConnection` 的 `connected` 方法内部会调用 `H` 的 `post` 方法向主线程发送消息，并且解决当前应用程序进程和 `Service` 跨进程通信的问题，在后面会详细介绍这一过程。在注释 5 处如果当前应用程序进程是第一个与 `Service` 进行绑定的，并且 `Service` 已经调用过 `onUnBind` 方法，则需要调用注释 6 处的代码。在注释 7 处如果应用程序进程的 `Client` 端没有发送过绑定 `Service` 的请求，则会调用注释 8 处的代码，注释 8 处和注释 6 处的代码区别就是最后一个参数 `rebind` 为 `false`，表示不是重新绑定。接着我们查看注释 6 处的 `requestServiceBindingLocked` 方法，代码如下所示：

```
frameworks/base/services/core/java/com/android/server/am/ActiveServices.java
```

```
private final boolean requestServiceBindingLocked(ServiceRecord r, IntentBindRecord i,
    boolean execInFg, boolean rebind) throws TransactionTooLargeException {
    ...
    if ((!i.requested || rebind) && i.apps.size() > 0) { //1
        try {
            bumpServiceExecutingLocked(r, execInFg, "bind");
            r.app.forceProcessStateUpTo(ActivityManager.PROCESS_STATE_SERVICE);
            r.app.thread.scheduleBindService(r, i.intent.getIntent(), rebind,
                r.app.repProcState); //2
            if (!rebind) {
                i.requested = true;
            }
            i.hasBound = true;
            i.doRebind = false;
        } catch (TransactionTooLargeException e) {
            ...
        }
        return true;
    }
}
```

注释 1 处 `i.requested` 表示是否发送过绑定 `Service` 的请求，从 `bindServiceLocked` 方法的注释 5 处得知是发送过的，因此，`!i.requested` 为 `false`。从 `bindServiceLocked` 方法的注释 5 处得知 `rebind` 值为 `true`，那么 `(!i.requested || rebind)` 的值为 `true`。`i.apps.size() > 0` 表示什么呢？其中 `i` 是 `IntentBindRecord` 类型的对象，AMS 会为每个绑定 `Service` 的 `Intent` 分配一个 `IntentBindRecord` 类型对象，代码如下所示：

```
frameworks/base/services/core/java/com/android/server/am/IntentBindRecord.java
```

```
final class IntentBindRecord {
    //被绑定的 Service
    final ServiceRecord service;
```

```

        //绑定 Service 的 Intent
        final Intent.FilterComparison intent;
        //所有用当前 Intent 绑定 Service 的应用程序进程
        final ArrayMap<ProcessRecord, AppBindRecord> apps
            = new ArrayMap<ProcessRecord, AppBindRecord>();//1
        ...
    }

```

我们来查看 IntentBindRecord 类，不同的应用程序进程可能使用同一个 Intent 来绑定 Service，因此在注释 1 处会用 apps 来存储所有用当前 Intent 绑定 Service 的应用程序进程。i.apps.size() > 0 表示所有用当前 Intent 绑定 Service 的应用程序进程个数大于 0，下面来验证 i.apps.size() > 0 是否为 true。我们回到 bindServiceLocked 方法的注释 1 处，ServiceRecord 的 retrieveAppBindingLocked 方法如下所示：

```

frameworks/base/services/core/java/com/android/server/am/ServiceRecord.java

public AppBindRecord retrieveAppBindingLocked(Intent intent,
        ProcessRecord app) {
    Intent.FilterComparison filter = new Intent.FilterComparison(intent);
    IntentBindRecord i = bindings.get(filter);
    if (i == null) {
        i = new IntentBindRecord(this, filter);//1
        bindings.put(filter, i);
    }
    AppBindRecord a = i.apps.get(app);//2
    if (a != null) {
        return a;
    }
    a = new AppBindRecord(this, i, app);//3
    i.apps.put(app, a);
    return a;
}

```

注释 1 处创建了 IntentBindRecord，注释 2 处根据 ProcessRecord 获得 IntentBindRecord 中存储的 AppBindRecord，如果 AppBindRecord 不为 null 就返回，如果为 null 就在注释 3 处创建 AppBindRecord，并将 ProcessRecord 作为 key，AppBindRecord 作为 value 保存在 IntentBindRecord 的 apps (i.apps) 中。回到 requestServiceBindingLocked 方法的注释 1 处，结合 ServiceRecord 的 retrieveAppBindingLocked 方法，我们得知 i.apps.size() > 0 为 true，这样就会调用注释 2 处的代码，r.app.thread 的类型为 IApplicationThread，它的实现我们已经很熟悉了，是 ActivityThread 的内部类 ApplicationThread，scheduleBindService 方法如下所示：

```
frameworks/base/core/java/android/app/ActivityThread.java
```

```
public final void scheduleBindService(IBinder token, Intent intent,
boolean rebind, int processState) {
    updateProcessState(processState, false);
    BindServiceData s = new BindServiceData();
    s.token = token;
    s.intent = intent;
    s.rebind = rebind;
    if (DEBUG_SERVICE)
        Slog.v(TAG, "scheduleBindService token=" + token + " intent=" + intent +
            " uid=" + Binder.getCallingUid() + " pid=" + Binder.getCallingPid());
    sendMessage(H.BIND_SERVICE, s);
}
```

首先将 Service 的信息封装成 BindServiceData 对象, BindServiceData 的成员变量 rebind 的值为 false, 后面会用到它。接着将 BindServiceData 传入到 sendMessage 方法中。sendMessage 向 H 发送消息, 我们接着查看 H 的 handleMessage 方法:

```
frameworks/base/core/java/android/app/ActivityThread.java
```

```
public void handleMessage(Message msg) {
    if (DEBUG_MESSAGES) Slog.v(TAG, ">>> handling: " + codeToString(msg.what));
    switch (msg.what) {
        ...
        case BIND_SERVICE:
            Trace.traceBegin(Trace.TRACE_TAG_ACTIVITY_MANAGER, "serviceBind");
            handleBindService((BindServiceData)msg.obj);
            Trace.traceEnd(Trace.TRACE_TAG_ACTIVITY_MANAGER);
            break;
        ...
    }
    ...
}
```

H 在接收到 BIND_SERVICE 类型消息时, 会在 handleMessage 方法中会调用 handleBindService 方法:

```
frameworks/base/core/java/android/app/ActivityThread.java
```

```
private void handleBindService(BindServiceData data) {
    Service s = mServices.get(data.token); //1
    if (DEBUG_SERVICE)
        Slog.v(TAG, "handleBindService s=" + s + " rebind=" + data.rebind);
    if (s != null) {
        try {
```

```

data.intent.setExtrasClassLoader(s.getClassLoader());
data.intent.prepareToEnterProcess();
try {
    if (!data.rebind) { //2
        IBinder binder = s.onBind(data.intent); //3
        ActivityManager.getService().publishService(
            data.token, data.intent, binder); //4
    } else {
        s.onRebind(data.intent); //5
        ActivityManager.getService().serviceDoneExecuting(
            data.token, SERVICE_DONE_EXECUTING_ANON, 0, 0);
    }
    ensureJitEnabled();
} catch (RemoteException ex) {
    throw ex.rethrowFromSystemServer();
}
} catch (Exception e) {
    if (!mInstrumentation.onException(s, e)) {
        throw new RuntimeException(
            "Unable to bind to service " + s
            + " with " + data.intent + ": " + e.toString(), e);
    }
}
}
}

```

在注释 1 处获取要绑定的 Service。注释 2 处的 BindServiceData 的成员变量 rebind 的值为 false，这样会调用注释 3 处的代码来调用 Service 的 onBind 方法，到这里 Service 处于绑定状态了。如果 rebind 的值为 true 就会调用注释 5 处的 Service 的 onRebind 方法，这一点结合前文的 bindServiceLocked 方法的注释 5 处，得出的结论就是：如果当前应用程序进程第一个与 Service 进行绑定，并且 Service 已经调用过 onUnBind 方法，则会调用 Service 的 onRebind 方法。handleBindService 方法有两个分支，一个是绑定过 Service 的情况，另一个是未绑定的情况，这里分析未绑定的情况，查看注释 4 处的代码，实际上是调用 AMS 的 publishService 方法。讲到这里，先给出这一部分的代码时序图（不包括 Service 启动过程），如图 4-10 所示。

接着来查看 AMS 的 publishService 方法，代码如下所示：

```
frameworks/base/services/core/java/com/android/server/am/ActivityManagerService.java
```

```

public void publishService(IBinder token, Intent intent, IBinder service) {
    if (intent != null && intent.hasFileDescriptors() == true) {
        throw new IllegalArgumentException("File descriptors passed in Intent");
    }
}

```

```

    }
    synchronized(this) {
        if (!(token instanceof ServiceRecord)) {
            throw new IllegalArgumentException("Invalid service token");
        }
        mServices.publishServiceLocked((ServiceRecord)token, intent, service);
    }
}

```

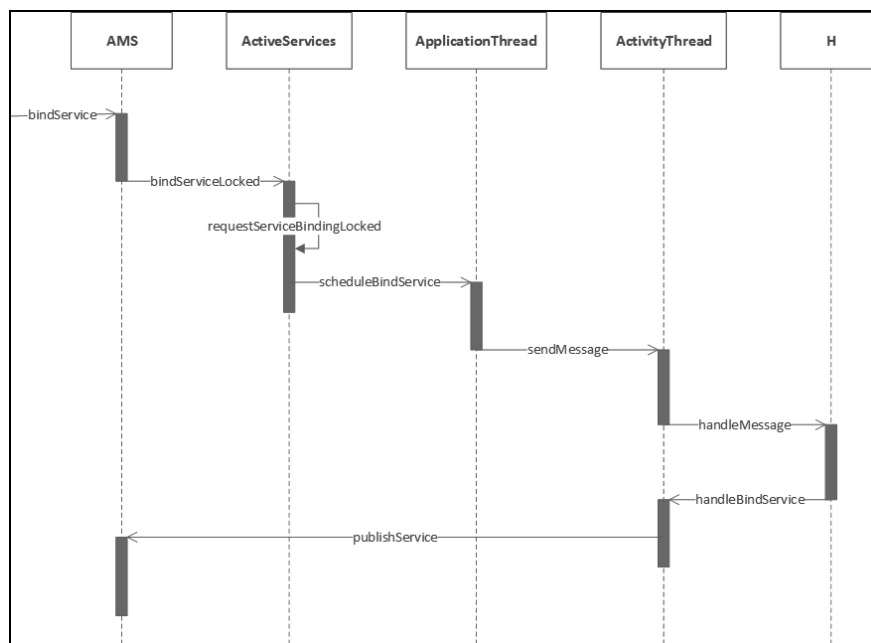


图 4-10 Service 的绑定过程前半部分调用关系时序图

在 publishService 方法中调用了 ActiveServices 类型的 mServices 对象的 publishServiceLocked 方法：

frameworks/base/services/core/java/com/android/server/am/ActiveServices.java

```

void publishServiceLocked(ServiceRecord r, Intent intent, IBinder service) {
    final long origId = Binder.clearCallingIdentity();
    try {
        ...
        for (int conni=r.connections.size()-1; conni>=0; conni--) {
            ...
            try {
                c.conn.connected(r.name, service, false); //1
            } catch (Exception e) {

```

```

        Slog.w(TAG, "Failure sending service " + r.name +
            " to connection " + c.conn.asBinder() +
            " (in " + c.binding.client.processName + ")", e);
    }
}
}
}
serviceDoneExecutingLocked(r, mDestroyingServices.contains(r), false);
}
} finally {
    Binder.restoreCallingIdentity(origId);
}
}

```

注释 1 处的代码在前面介绍过，c.conn 指的是 IServiceConnection，它是 ServiceConnection 在本地的代理，用于解决当前应用程序进程和 Service 跨进程通信的问题，具体实现为 ServiceDispatcher.InnerConnection，其中 ServiceDispatcher 是 LoadedApk 的内部类，ServiceDispatcher.InnerConnection 的 connected 方法的代码如下所示：

frameworks/base/core/java/android/app/LoadedApk.java

```

static final class ServiceDispatcher {
    ...
    private static class InnerConnection extends IServiceConnection.Stub {
        final WeakReference<LoadedApk.ServiceDispatcher> mDispatcher;
        InnerConnection(LoadedApk.ServiceDispatcher sd) {
            mDispatcher = new WeakReference<LoadedApk.ServiceDispatcher>(sd);
        }
        public void connected(ComponentName name, IBinder service) throws
            RemoteException {
            LoadedApk.ServiceDispatcher sd = mDispatcher.get();
            if (sd != null) {
                sd.connected(name, service);//1
            }
        }
    }
    ...
}

```

在注释 1 处调用了 ServiceDispatcher 类型的 sd 对象的 connected 方法，代码如下所示：

frameworks/base/core/java/android/app/LoadedApk.java

```

public void connected(ComponentName name, IBinder service, boolean dead) {
    if (mActivityThread != null) {

```

```

        mActivityThread.post(new RunConnection(name, service, 0, dead)); //1
    } else {
        doConnected(name, service, dead);
    }
}

```

在注释 1 处调用 Handler 类型的对象 mActivityThread 的 post 方法, mActivityThread 实际上指向的是 H。因此,通过调用 H 的 post 方法将 RunConnection 对象的内容运行在主线程中。RunConnection 是 LoadedApk 的内部类,定义如下所示:

frameworks/base/core/java/android/app/LoadedApk.java

```

private final class RunConnection implements Runnable {
    RunConnection(ComponentName name, IBinder service, int command, boolean
    dead) {
        mName = name;
        mService = service;
        mCommand = command;
        mDead = dead;
    }
    public void run() {
        if (mCommand == 0) {
            doConnected(mName, mService, mDead);
        } else if (mCommand == 1) {
            doDeath(mName, mService);
        }
    }
    final ComponentName mName;
    final IBinder mService;
    final int mCommand;
    final boolean mDead;
}

```

在 RunConnection 的 run 方法中调用了 doConnected 方法:

frameworks/base/core/java/android/app/LoadedApk.java

```

public void doConnected(ComponentName name, IBinder service, boolean dead) {
    ...
    if (old != null) {
        mConnection.onServiceDisconnected(name);
    }
    if (dead) {
        mConnection.onBindingDied(name);
    }
    if (service != null) {

```

```

        mConnection.onServiceConnected(name, service); //1
    }

```

在注释 1 处调用了 ServiceConnection 类型的对象 mConnection 的 onServiceConnected 方法，这样在客户端实现了 ServiceConnection 接口类的 onServiceConnected 方法就会被执行。至此，Service 的绑定过程就分析完成。最后给出剩余部分的代码时序图，如图 4-11 所示。

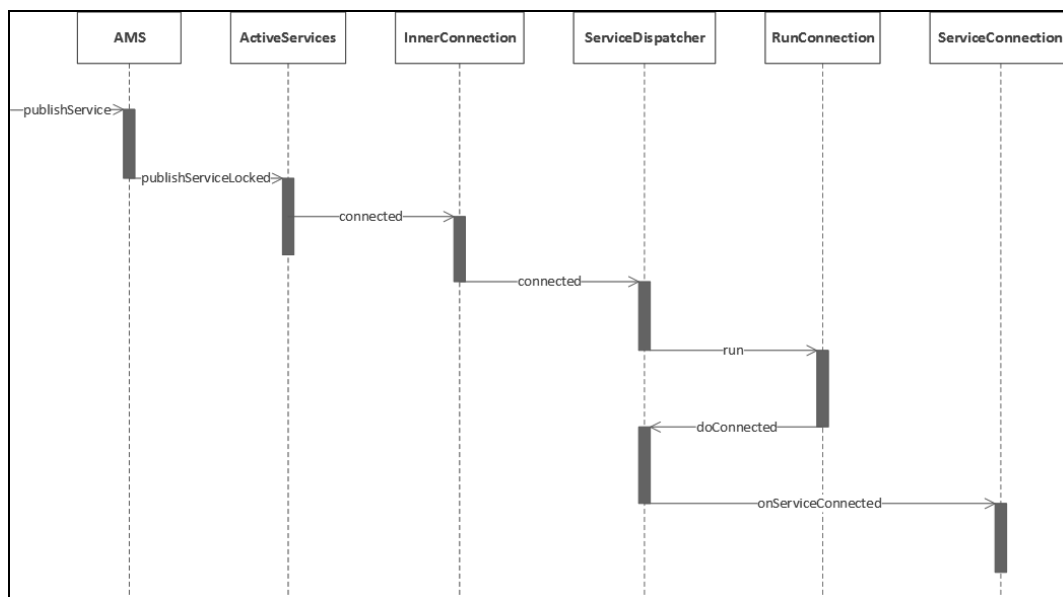


图 4-11 Service 的绑定过程剩余部分的代码时序图

4.4 广播的注册、发送和接收过程

广播作为四大组件之一，使用频率远没有 Activity 高，但是广播的工作过程还是十分有必要了解的。本节主要从三个方面讲解广播工作过程，分别是广播的注册、发送和接收，这些过程和本章前 3 节重叠的部分，这一节不会再赘述，而是一笔带过，建议阅读本节前先阅读本章前面 3 节的内容。

4.4.1 广播的注册过程

广播的注册通俗来讲就是广播接收者注册自己感兴趣的广播，广播的注册分为两种，

分别是静态注册和动态注册，静态注册在应用安装时由 PackageManagerService 来完成注册过程，关于这一过程本节不做介绍，这里只介绍广播的动态注册，时序图如图 4-12 所示。

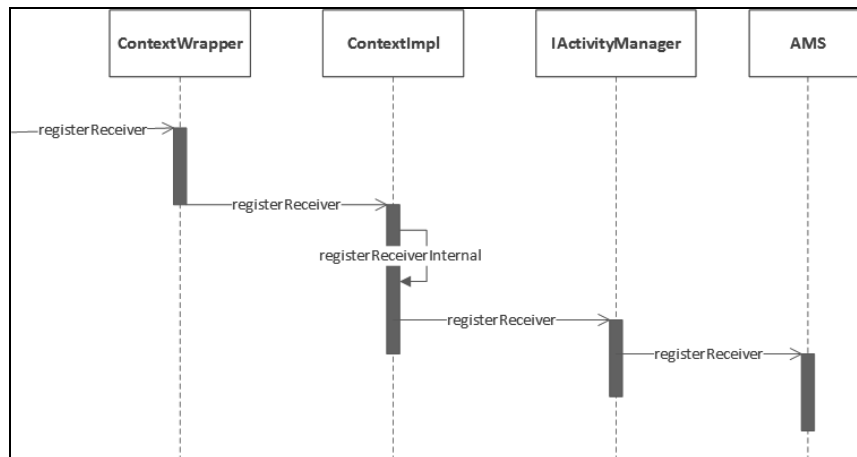


图 4-12 广播的动态注册过程时序图

要想动态注册广播，需要调用 registerReceiver 方法，它在 ContextWrapper 中实现，代码如下所示：

frameworks/base/core/java/android/content/ContextWrapper.java

```

@Override
public Intent registerReceiver(
    BroadcastReceiver receiver, IntentFilter filter,
    String broadcastPermission, Handler scheduler) {
    return mBase.registerReceiver(receiver, filter, broadcastPermission,
        scheduler);
}

```

这里 mBase 具体指向就是 ContextImpl，ContextImpl 的 registerReceiver 方法有很多重载的方法最终会调用 registerReceiverInternal 方法：

frameworks/base/core/java/android/app/ContextImpl.java

```

private Intent registerReceiverInternal(BroadcastReceiver receiver, int userId,
    IntentFilter filter, String broadcastPermission,
    Handler scheduler, Context context, int flags) {
    IIntentReceiver rd = null;
    if (receiver != null) {
        if (mPackageInfo != null && context != null) { //1
            if (scheduler == null) {

```

```

        scheduler = mMainThread.getHandler();
    }
    rd = mPackageInfo.getReceiverDispatcher(
        receiver, context, scheduler,
        mMainThread.getInstrumentation(), true); //2
    } else {
        if (scheduler == null) {
            scheduler = mMainThread.getHandler();
        }
        rd = new LoadedApk.ReceiverDispatcher(receiver, context, scheduler,
            null, true).getIIntentReceiver(); //3
    }
}
try {
    final Intent intent = ActivityManager.getService().registerReceiver
        (mMainThread.getApplicationThread(), mBasePackageName, rd, filter,
        broadcastPermission, userId, flags); //4
    if (intent != null) {
        intent.setExtrasClassLoader(getClassLoader());
        intent.prepareToEnterProcess();
    }
    return intent;
} catch (RemoteException e) {
    throw e.rethrowFromSystemServer();
}
}

```

在注释 1 处判断如果 LoadedApk 类型的 mPackageInfo 不等于 null，并且 context 不等于 null 就调用注释 2 处的代码，通过 mPackageInfo 的 getReceiverDispatcher 方法获取 rd 对象，否则就调用注释 3 处的代码来创建 rd 对象。注释 2 和注释 3 处的代码的目的都是要获取 IIntentReceiver 类型的 rd 对象，IIntentReceiver 是一个 Binder 接口，用于广播的跨进程的通信，它在 LoadedApk.ReceiverDispatcher.InnerReceiver 中实现，如下所示：

frameworks/base/core/java/android/app/LoadedApk.java

```

static final class ReceiverDispatcher {
    final static class InnerReceiver extends IIntentReceiver.Stub {
        final WeakReference<LoadedApk.ReceiverDispatcher> mDispatcher;
        final LoadedApk.ReceiverDispatcher mStrongRef;
        InnerReceiver(LoadedApk.ReceiverDispatcher rd, boolean strong) {
            mDispatcher = new WeakReference<LoadedApk.ReceiverDispatcher>(rd);
            mStrongRef = strong ? rd : null;
        }
    }
    ...
}

```

```

    }
    ...
}

```

回到 `registerReceiverInternal` 方法,在注释4处调用了 `IActivityManager` 的 `registerReceiver` 方法,最终会调用 AMS 的 `registerReceiver` 方法,并将 `IIntentReceiver` 类型的 `rd` 传进去,这里之所以不直接传入 `BroadcastReceiver` 而是传入 `IIntentReceiver`,是因为注册广播是一个跨进程过程,需要具有跨进程的通信功能的 `IIntentReceiver`。`registerReceiver` 方法内容比较多,这里分为两个部分来进行讲解,先来查看 `registerReceiver` 方法的 part1,如下所示:

1. registerReceiver 方法的 part1

frameworks/base/services/core/java/com/android/server/am/ActivityManagerService.java

```

public Intent registerReceiver(IApplicationThread caller, String callerPackage,
    IIntentReceiver receiver, IntentFilter filter, String permission, int userId,
    int flags) {
    ...
    ProcessRecord callerApp = null;
    ...
    synchronized(this) {
        if (caller != null) {
            callerApp = getRecordForAppLocked(caller); //1
            ...
            Iterator<String> actions = filter.actionsIterator(); //2
            if (actions == null) {
                ArrayList<String> noAction = new ArrayList<String>(1);
                noAction.add(null);
                actions = noAction.iterator();
            }
            int[] userIds = { UserHandle.USER_ALL, UserHandle.getUserId(callingUid) };
            while (actions.hasNext()) {
                String action = actions.next();
                for (int id : userIds) {
                    ArrayList<Intent> stickies = mStickyBroadcasts.get(id);
                    if (stickies != null) {
                        ArrayList<Intent> intents = stickies.get(action);
                        if (intents != null) {
                            if (stickyIntents == null) {
                                stickyIntents = new ArrayList<Intent>();
                            }
                            stickyIntents.addAll(intents); //3
                        }
                    }
                }
            }
        }
    }
}

```

```

    }
    }
}

ArrayList<Intent> allSticky = null;
if (stickyIntents != null) {
    final ContentResolver resolver = mContext.getContentResolver();
    //遍历寻找匹配的粘性广播
    for (int i = 0, N = stickyIntents.size(); i < N; i++) {
        Intent intent = stickyIntents.get(i);
        if (instantApp && (intent.getFlags()
            & Intent.FLAG_RECEIVER_VISIBLE_TO_INSTANT_APPS) == 0) {
            continue;
        }
        if (filter.match(resolver, intent, true, TAG) >= 0) {
            if (allSticky == null) {
                allSticky = new ArrayList<Intent>();
            }
            allSticky.add(intent);//4
        }
    }
}
...
}

```

在注释1处通过 `getRecordForAppLocked` 方法得到 `ProcessRecord` 类型的 `callerApp` 对象，它用于描述请求 AMS 注册广播接收者的 Activity 所在的应用程序进程。在注释2处根据传入的 `IntentFilter` 类型 `filter` 得到 `actions` 列表，根据 `actions` 列表和 `userIds` (`userIds` 可以理解为应用程序的 `uid`) 得到所有的粘性广播的 `intent`，并在注释3处传入到 `stickyIntents` 中。接下来从 `stickyIntents` 中找到匹配传入的参数 `filter` 的粘性广播的 `intent`，在注释4处将这些 `intent` 存入到 `allSticky` 列表中，从这里可以看出粘性广播是存储在 AMS 中的。

2. registerReceiver 方法的 part2

接下来查看 AMS 的 `registerReceiver` 方法的剩余内容，如下所示：

`frameworks/base/services/core/java/com/android/server/am/ActivityManagerService.java`

```

public Intent registerReceiver(IApplicationThread caller, String callerPackage,
    IIntentReceiver receiver, IntentFilter filter, String permission, int
    userId) {
    ...
    synchronized (this) {

```

```

...
ReceiverList rl = mRegisteredReceivers.get(receiver.asBinder()); //1
if (rl == null) {
    rl = new ReceiverList(this, callerApp, callingPid, callingUid,
        userId, receiver); //2
    if (rl.app != null) {
        rl.app.receivers.add(rl);
    }
    ...
}
...
BroadcastFilter bf = new BroadcastFilter(filter, rl, callerPackage,
    permission, callingUid, userId); //3
rl.add(bf); //4
if (!bf.debugCheck()) {
    Slog.w(TAG, "==> For Dynamic broadcast");
}
mReceiverResolver.addFilter(bf); //5
...
return sticky;
}
}

```

在注释 1 处获取 ReceiverList 列表，如果为空则在注释 2 处创建，ReceiverList 继承自 ArrayList，用来存储广播接收者。在注释 3 处创建 BroadcastFilter 并传入此前创建的 ReceiverList，BroadcastFilter 用来描述注册的广播接收者，并在注释 4 处通过 add 方法将自身添加到 ReceiverList 中。在注释 5 处将 BroadcastFilter 添加到 IntentResolver 类型的 mReceiverResolver 中，这样当 AMS 接收到广播时就可以从 mReceiverResolver 中找到对应的广播接收者了，从而达到了注册广播的目的。

4.4.2 广播的发送和接收过程

广播的发送和接收过程分为两个部分来进行讲解，分别是 ContextImpl 到 AMS 的调用过程和 AMS 到 BroadcastReceiver 的调用过程。

4.4.2.1 ContextImpl 到 AMS 的调用过程

广播可以发送多种类型，包括无序广播（普通广播）、有序广播和粘性广播，这里以无序广播为例来讲解广播的发送过程。要发送无序广播需要调用 sendBroadcast 方法，它同样在 ContextWrapper 中实现，按照惯例先给出 ContextImpl 到 AMS 的调用过程的时序图，如图 4-13 所示。

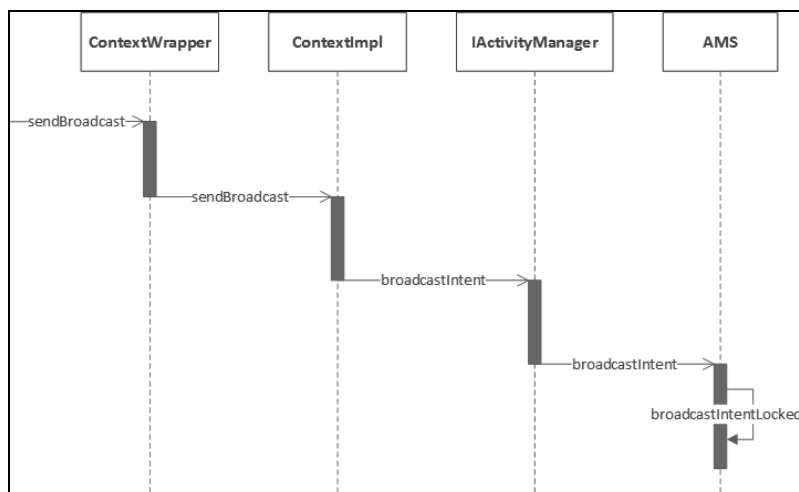


图 4-13 ContextImpl 到 AMS 的调用过程的时序图

先来查看 ContextWrapper 的 sendBroadcast 方法，如下所示：

frameworks/base/core/java/android/content/ContextWrapper.java

```

@Override
public void sendBroadcast(Intent intent) {
    mBase.sendBroadcast(intent);
}

```

接着来看 ContextImpl 中的 sendBroadcast 方法，如下所示：

frameworks/base/core/java/android/app/ContextImpl.java

```

@Override
public void sendBroadcast(Intent intent) {
    warnIfCallingFromSystemProcess();
    String resolvedType = intent.resolveTypeIfNeeded(getContentResolver());
    try {
        intent.prepareToLeaveProcess(this);
        ActivityManager.getService().broadcastIntent(
            mMainThread.getApplicationThread(), intent, resolvedType, null,
            Activity.RESULT_OK, null, null, null, AppOpsManager.OP_NONE, null,
            false, false, getUserId());
    } catch (RemoteException e) {
        throw e.rethrowFromSystemServer();
    }
}

```

又是熟悉的代码，最终会调用 AMS 的 broadcastIntent 方法：

```
frameworks/base/services/core/java/com/android/server/am/ActivityManagerService.java
```

```
public final int broadcastIntent(IApplicationThread caller,
    Intent intent, String resolvedType, IIntentReceiver resultTo,
    int resultCode, String resultData, Bundle resultExtras,
    String[] requiredPermissions, int appOp, Bundle bOptions,
    boolean serialized, boolean sticky, int userId) {
    enforceNotIsolatedCaller("broadcastIntent");
    synchronized(this) {
        //验证广播是否合法
        intent = verifyBroadcastLocked(intent);;//1
        final ProcessRecord callerApp = getRecordForAppLocked(caller);
        final int callingPid = Binder.getCallingPid();
        final int callingUid = Binder.getCallingUid();
        final long origId = Binder.clearCallingIdentity();
        /**
         * 2
         */
        int res = broadcastIntentLocked(callerApp, callerApp != null ?
            callerApp.info.packageName : null, intent, resolvedType, resultTo,
            resultCode, resultData, resultExtras, requiredPermissions, appOp, bOptions,
            serialized, sticky, callingPid, callingUid, userId);
        Binder.restoreCallingIdentity(origId);
        return res;
    }
}
```

我们先来查看注释 1 处的 verifyBroadcastLocked 方法:

```
frameworks/base/services/core/java/com/android/server/am/ActivityManagerService.java
```

```
final Intent verifyBroadcastLocked(Intent intent) {
    if (intent != null && intent.hasFileDescriptors() == true) {//1
        throw new IllegalArgumentException("File descriptors passed in Intent");
    }
    int flags = intent.getFlags();//2
    if (!mProcessesReady) {
        if ((flags&Intent.FLAG_RECEIVER_REGISTERED_ONLY_BEFORE_BOOT) != 0) {//3
        } else if ((flags&Intent.FLAG_RECEIVER_REGISTERED_ONLY) == 0) {//4
            Slog.e(TAG, "Attempt to launch receivers of broadcast intent " + intent
                + " before boot completion");
            throw new IllegalStateException("Cannot broadcast before boot
                completed");
        }
    }
}
...
}
```

```

        return intent;
    }

```

verifyBroadcastLocked 方法主要验证广播是否合法，在注释 1 处验证 intent 是否不为 null 并且有文件描述符。注释 2 处获得 intent 中的 flag。注释 3 处如果系统正在启动过程中，判断如果 flag 设置为 FLAG_RECEIVER_REGISTERED_ONLY_BEFORE_BOOT(启动检查时只接受动态注册的广播接收者) 则不做处理，如果不是则在注释 4 处判断如果 flag 没有设置为 FLAG_RECEIVER_REGISTERED_ONLY（只接受动态注册的广播接收者）则会抛出异常。我们再回到 broadcastIntent 方法，在注释 2 处调用了 broadcastIntentLocked 方法，代码如下所示：

frameworks/base/services/core/java/com/android/server/am/ActivityManagerService.java

```

final int broadcastIntentLocked(ProcessRecord callerApp,
    String callerPackage, Intent intent, String resolvedType,
    IIntentReceiver resultTo, int resultCode, String resultData,
    Bundle resultExtras, String[] requiredPermissions, int appOp, Bundle
    bOptions, boolean ordered, boolean sticky, int callingPid,
    int callingUid, int userId) {
    ...
    if ((receivers != null && receivers.size() > 0)
        || resultTo != null) {
        BroadcastQueue queue = broadcastQueueForIntent(intent);
        /**
         * 1
         */
        BroadcastRecord r = new BroadcastRecord(queue, intent, callerApp,
            callerPackage, callingPid, callingUid, resolvedType, requiredPermissions,
            appOp, bOptions, receivers, resultTo, resultCode, resultData,
            resultExtras, ordered, sticky, false, userId);
        ...
        boolean replaced = replacePending && queue.replaceOrderedBroadcastLocked(r);
        if (!replaced) {
            queue.enqueueOrderedBroadcastLocked(r);
            queue.scheduleBroadcastsLocked();//2
        }
    }
    ...
}
return ActivityManager.BROADCAST_SUCCESS;
}

```

这里省略了很多代码，前面的工作主要是将动态注册的广播接收者和静态注册的广播

接收者按照优先级高低不同存储在不同的列表中,再将这两个列表合并到 receivers 列表中,这样 receivers 列表包含了所有的广播接收者(无序广播和有序广播)。在注释 1 处创建 BroadcastRecord 对象并将 receivers 传进去,在注释 2 处调用 BroadcastQueue 的 scheduleBroadcastsLocked 方法。

4.4.2.2 AMS 到 BroadcastReceiver 的调用过程

AMS 到 BroadcastReceiver 的调用过程的时序图如图 4-14 所示。

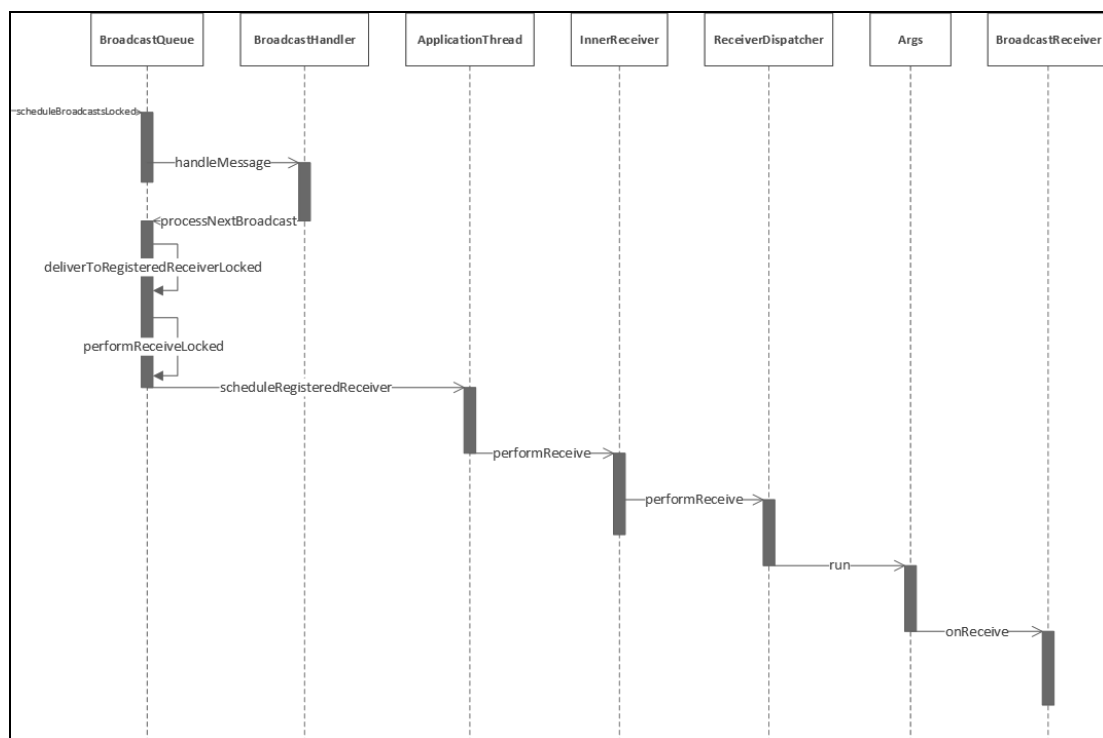


图 4-14 AMS 到 BroadcastReceiver 的调用过程的时序图

BroadcastQueue 的 scheduleBroadcastsLocked 方法的代码如下所示:

```

public void scheduleBroadcastsLocked() {
    if (DEBUG_BROADCAST) Slog.v(TAG_BROADCAST, "Schedule broadcasts ["
        + mQueueName + "]: current="
        + mBroadcastsScheduled);

    if (mBroadcastsScheduled) {
        return;
    }
}

```

```

    }
    mHandler.sendMessage(mHandler.obtainMessage(BROADCAST_INTENT_MSG,
        this)); //1
    mBroadcastsScheduled = true;
}

```

在注释1处向BroadcastHandler类型的mHandler对象发送了BROADCAST_INTENT_MSG类型的消息，这个消息在BroadcastHandler的handleMessage方法中进行处理，如下所示：

frameworks/base/services/core/java/com/android/server/am/BroadcastQueue.java

```

private final class BroadcastHandler extends Handler {
    public BroadcastHandler(Looper looper) {
        super(looper, null, true);
    }

    @Override
    public void handleMessage(Message msg) {
        switch (msg.what) {
            case BROADCAST_INTENT_MSG: {
                if (DEBUG_BROADCAST) Slog.v(
                    TAG_BROADCAST, "Received BROADCAST_INTENT_MSG");
                processNextBroadcast(true);
            } break;
            case BROADCAST_TIMEOUT_MSG: {
                synchronized (mService) {
                    broadcastTimeoutLocked(true);
                }
            } break;
        }
    }
}

```

在handleMessage方法中调用了processNextBroadcast方法，方法对无序广播和有序广播分别进行处理，旨在将广播发送给广播接收者，下面给出processNextBroadcast方法中对无序广播的处理部分：

frameworks/base/services/core/java/com/android/server/am/BroadcastQueue.java

```

final void processNextBroadcast(boolean fromMsg) {
    ...
    if (fromMsg) {
        //已经处理了 BROADCAST_INTENT_MSG 类型的消息
        mBroadcastsScheduled = false; //1
    }
}

```

```

//遍历存储无序广播的 mParallelBroadcasts 列表
while (mParallelBroadcasts.size() > 0) { //2
    //获取无序广播
    r = mParallelBroadcasts.remove(0); //3
    ...
    for (int i=0; i<N; i++) {
        Object target = r.receivers.get(i);
        if (DEBUG_BROADCAST) Slog.v(TAG_BROADCAST,
            "Delivering non-ordered on [" + mQueueName + "] to registered "
            + target + ": " + r);
        deliverToRegisteredReceiverLocked(r, (BroadcastFilter)target, false,
            i); //4
    }
    ...
}
}

```

从前面 `BroadcastHandler` 方法中我们得知传入的参数 `fromMsg` 的值为 `true`，因此在注释 1 处将 `mBroadcastsScheduled` 设置为 `false`，表示对于此前发来的 `BROADCAST_INTENT_MSG` 类型的消息已经处理了。注释 2 处的 `mParallelBroadcasts` 列表用来存储无序广播，通过 `while` 循环将 `mParallelBroadcasts` 列表中的无序广播发送给对应的广播接收者。在注释 3 处获取每一个 `mParallelBroadcasts` 列表中存储的 `BroadcastRecord` 类型的 `r` 对象。在注释 4 处将这些 `r` 对象描述的广播发送给对应的广播接收者，`deliverToRegisteredReceiverLocked` 方法如下所示：

```

frameworks/base/services/core/java/com/android/server/am/BroadcastQueue.java

private void deliverToRegisteredReceiverLocked(BroadcastRecord r,
    BroadcastFilter filter, boolean ordered, int index) {
    ...
    try {
        if (DEBUG_BROADCAST_LIGHT) Slog.i(TAG_BROADCAST,
            "Delivering to " + filter + " : " + r);
        if (filter.receiverList.app != null && filter.receiverList.app.
            inFullBackup) {
            ...
        } else {
            performReceiveLocked(filter.receiverList.app,
                filter.receiverList.receiver, new Intent(r.intent), r.resultCode,
                r.resultData, r.resultExtras, r.ordered, r.initialSticky,
                r.userId); //1
        }
        if (ordered) {

```

```

        r.state = BroadcastRecord.CALL_DONE_RECEIVE;
    }
    } catch (RemoteException e) {
    ...
    }
}

```

这里省去了大部分的代码，这些代码是用来检查广播发送者和广播接收者的权限的。如果通过了权限的检查，则会调用注释 1 处的 `performReceiveLocked` 方法：

frameworks/base/services/core/java/com/android/server/am/BroadcastQueue.java

```

void performReceiveLocked(ProcessRecord app, IIntentReceiver receiver,
    Intent intent, int resultCode, String data, Bundle extras,
    boolean ordered, boolean sticky, int sendingUser) throws RemoteException {
    if (app != null) { //1
        if (app.thread != null) { //2
            try {
                app.thread.scheduleRegisteredReceiver(receiver, intent, resultCode,
                    data, extras, ordered, sticky, sendingUser, app.repProcState); //3
            } catch (RemoteException ex) {
                ...
            }
        } else {
            throw new RemoteException("app.thread must not be null");
        }
    } else {
        receiver.performReceive(intent, resultCode, data, extras, ordered,
            sticky, sendingUser);
    }
}

```

在注释 1 和注释 2 处的代码表示如果广播接收者所在的应用程序进程存在并且正在运行，则执行注释 3 处的代码，表示用广播接收者所在的应用程序进程来接收广播，这里 `app.thread` 指的是 `ApplicationThread`，我们来查看 `ApplicationThread` 的 `scheduleRegisteredReceiver` 方法，代码如下所示：

frameworks/base/core/java/android/app/ActivityThread.java

```

public void scheduleRegisteredReceiver(IIntentReceiver receiver, Intent intent,
    int resultCode, String dataStr, Bundle extras, boolean ordered,
    boolean sticky, int sendingUser, int processState) throws RemoteException {
    updateProcessState(processState, false);
    receiver.performReceive(intent, resultCode, dataStr, extras, ordered,
        sticky, sendingUser);
}

```

在 `scheduleRegisteredReceiver` 方法中调用了 `IIntentReceiver` 类型的对象 `receiver` 的 `performReceive` 方法，`IIntentReceiver` 在前面提到过，用于广播的跨进程的通信，它的具体实现为 `LoadedApk.ReceiverDispatcher.InnerReceiver`，代码如下所示：

`frameworks/base/core/java/android/app/LoadedApk.java`

```
static final class ReceiverDispatcher {
    final static class InnerReceiver extends IIntentReceiver.Stub {
        final WeakReference<LoadedApk.ReceiverDispatcher> mDispatcher;
        final LoadedApk.ReceiverDispatcher mStrongRef;
        InnerReceiver(LoadedApk.ReceiverDispatcher rd, boolean strong) {
            mDispatcher = new WeakReference<LoadedApk.ReceiverDispatcher>(rd);
            mStrongRef = strong ? rd : null;
        }
        @Override
        public void performReceive(Intent intent, int resultCode, String data,
            Bundle extras, boolean ordered, boolean sticky, int sendingUser) {
            final LoadedApk.ReceiverDispatcher rd;
            if (intent == null) {
                Log.wtf(TAG, "Null intent received");
                rd = null;
            } else {
                rd = mDispatcher.get();
            }
            if (ActivityThread.DEBUG_BROADCAST) {
                int seq = intent.getIntExtra("seq", -1);
                Slog.i(ActivityThread.TAG, "Receiving broadcast " + intent.
                    getAction() + " seq=" + seq + " to " +
                    (rd != null ? rd.mReceiver : null));
            }
            if (rd != null) {
                rd.performReceive(intent, resultCode, data, extras,
                    ordered, sticky, sendingUser);//1
            } else {
                ...
            }
        }
    }
    ...
}
```

`IIntentReceiver` 和 `IActivityManager` 一样，都使用了 AIDL 来实现进程间通信。`InnerReceiver` 继承自 `IIntentReceiver.Stub`，是 Binder 通信的服务器端，`IIntentReceiver` 则是 Binder 通信的客户端、`InnerReceiver` 在本地的代理，它的具体实现就是 `InnerReceiver`。在 `InnerReceiver` 的 `performReceive` 方法的注释 1 处调用了 `ReceiverDispatcher` 类型的 `rd` 对

象的 performReceive 方法，如下所示：

frameworks/base/core/java/android/app/LoadedApk.java

```
public void performReceive(Intent intent, int resultCode, String data,
    Bundle extras, boolean ordered, boolean sticky, int sendingUser) {
    final Args args = new Args(intent, resultCode, data, extras, ordered,
        sticky, sendingUser); //1
    if (intent == null) {
        Log.wtf(TAG, "Null intent received");
    } else {
        if (ActivityThread.DEBUG_BROADCAST) {
            int seq = intent.getIntExtra("seq", -1);
            Slog.i(ActivityThread.TAG, "Enqueueing broadcast " + intent.
                getAction()+ " seq=" + seq + " to " + mReceiver);
        }
    }
    if (intent == null || !mActivityThread.post(args.getRunnable())) { //2
        if (mRegistered && ordered) {
            IActivityManager mgr = ActivityManager.getService();
            if (ActivityThread.DEBUG_BROADCAST) Slog.i(ActivityThread.TAG,
                "Finishing sync broadcast to " + mReceiver);
            args.sendFinished(mgr);
        }
    }
}
```

在注释 1 处将广播的 intent 等信息封装为 Args 对象，在注释 2 处调用 mActivityThread 的 post 方法并传入了 Args 对象。这个 mActivityThread 是一个 Handler 对象，具体指向的就是 H，注释 2 处的代码就是将 Args 对象的 getRunnable 方法通过 H 发送到线程的消息队列中，Args 的 getRunnable 方法如下所示：

frameworks/base/core/java/android/app/LoadedApk.java

```
final class Args extends BroadcastReceiver.PendingResult {
    ...
    public final Runnable getRunnable() {
        return () -> {
            try {
                ClassLoader cl = mReceiver.getClass().getClassLoader();
                intent.setExtrasClassLoader(cl);
                intent.prepareToEnterProcess();
                setExtrasClassLoader(cl);
                receiver.setPendingResult(this);
                receiver.onReceive(mContext, intent); //1
            }
        }
    }
}
```

```

        } catch (Exception e) {
            ...
        }
        ...
    }
}

```

在注释 1 处执行了 BroadcastReceiver 类型的 receiver 对象的 onReceive 方法, 这样注册的广播接收者就收到了广播并得到了 intent。

4.5 Content Provider的启动过程

Content Provider 作为四大组件之一, 即内容提供者, 在通常情况下并没有其他的组件使用频繁, 主要用于进程内和进程间的数据共享。Content Provider 的启动过程分为两个部分来进行讲解, 分别是 query 方法到 AMS 的调用过程和 AMS 启动 Content Provider 的过程。

4.5.1 query方法到AMS的调用过程

为了便于理解 Content Provider 的启动过程, 首先列出一段使用 Content Provider 的代码, 如下所示:

```

public class ContentProviderActivity extends AppCompatActivity {
    private final static String TAG = "ContentProviderActivity";
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_content_provider);
        Uri uri = Uri.parse("content://com.example.liuwangshu.mooncontentprovide.
        GameProvider");
        ContentValues mContentValues = new ContentValues();
        mContentValues.put("_id", 2);
        mContentValues.put("name", "大航海时代 01");
        mContentValues.put("describe", "最好玩的航海网游");
        getContentResolver().insert(uri, mContentValues); //1
    }
}

```

在 ContentProviderActivity 中, 我们在 GameProvider 中插入了一条数据, 可见要想插入一条数据, 或者说使用 ContentProvider, 需要先调用 getContentResolver 方法, 如下所示:

```
frameworks/base/core/Java/android/content/ContextWrapper.java
```

```
@Override
public ContentResolver getContentResolver() {
    return mBase.getContentResolver();
}
```

这里的 mBase 已经是我们的老朋友了，指的是 ContextImpl，ContextImpl 的 getContentResolver 方法如下所示：

```
frameworks/base/core/java/android/app/ContextImpl.java
```

```
@Override
public ContentResolver getContentResolver() {
    return mContentResolver;
}
```

getContentResolver 方法中返回了 ApplicationContentResolver 类型的 mContentResolver 对象，ApplicationContentResolver 是 ContextImpl 中的静态内部类，继承自 ContentResolver，它在 ContextImpl 的构造方法中被创建，这说明当我们调用 ContentResolver 的 insert、query、update 等方法时就会启动 Content Provider。这里以 query 方法来进行举例，query 方法在 ApplicationContentResolver 的父类 ContentResolver 中实现，有 3 个重载方法，最终会调用如下的 query 方法：

```
frameworks/base/core/java/android/content/ContentResolver.java
```

```
public final @Nullable Cursor query(final @RequiresPermission.Read @NonNull Uri uri,
    @Nullable String[] projection, @Nullable Bundle queryArgs,
    @Nullable CancellationSignal cancellationSignal) {
    Preconditions.checkNotNull(uri, "uri");
    IContentProvider unstableProvider = acquireUnstableProvider(uri);//1
    ...
    try {
        ...
    }
    try {
        qCursor = unstableProvider.query(mPackageName, uri, projection,
            queryArgs, remoteCancellationSignal);//2
    } catch (DeadObjectException e) {
        ...
    }
    ...
} catch (RemoteException e) {
    return null;
} finally {
```



```

    ...
}
}

```

在注释 1 处通过 `acquireUnstableProvider` 方法返回 `IContentProvider` 类型的 `unstableProvider` 对象，在注释 2 处调用 `unstableProvider` 的 `query` 方法。`IContentProvider` 是 `ContentProvider` 在本地的代理，具体的实现为 `ContentProvider`，我们查看 `ContentProvider` 的 `acquireUnstableProvider` 方法做了什么，如下所示：

```
frameworks/base/core/java/android/content/ContentResolver.java
```

```

public final IContentProvider acquireUnstableProvider(Uri uri) {
    if (!SCHEME_CONTENT.equals(uri.getScheme())) { //1
        return null;
    }
    String auth = uri.getAuthority();
    if (auth != null) {
        return acquireUnstableProvider(mContext, uri.getAuthority()); //2
    }
    return null;
}

```

在注释 1 处检查 `uri` 的 `scheme` 是否等于“content”（`SCHEME_CONTENT` 的值为“content”），如果不是则返回 `null`。在注释 2 处调用了 `acquireUnstableProvider` 方法，这是个抽象方法，它在 `ContentResolver` 的子类 `ApplicationContentResolver` 中实现，`ApplicationContentResolver` 是 `ContextImpl` 的静态内部类，如下所示：

```
frameworks/base/core/java/android/app/ContextImpl.java
```

```

private static final class ApplicationContentResolver extends ContentResolver {
    private final ActivityThread mMainThread;
    ...
    @Override
    protected IContentProvider acquireUnstableProvider(Context c, String auth) {
        return mMainThread.acquireProvider(c,
            ContentProvider.getAuthorityWithoutUserId(auth),
            resolveUserIdFromAuthority(auth), false);
    }
    ...
}

```

在 `acquireUnstableProvider` 方法中返回了 `ActivityThread` 类型的 `mMainThread` 对象的 `acquireProvider` 方法：

frameworks/base/core/java/android/app/ActivityThread.java

```

public final IContentProvider acquireProvider(
    Context c, String auth, int userId, boolean stable) {
    final IContentProvider provider = acquireExistingProvider(c, auth, userId,
stable);//1
    if (provider != null) {
        return provider;
    }
    ContentProviderHolder holder = null;
    try {
        holder = ActivityManager.getService().getContentProvider(
getApplicationThread(), auth, userId, stable);//2
    } catch (RemoteException ex) {
        throw ex.rethrowFromSystemServer();
    }
    if (holder == null) {
        Slog.e(TAG, "Failed to find provider info for " + auth);
        return null;
    }
    holder = installProvider(c, holder, holder.info,
true /*noisy*/, holder.noReleaseNeeded, stable);//3
    return holder.provider;
}

```

注释 1 处的 `acquireExistingProvider` 方法内部会检查 `ActivityThread` 的全局变量 `mProviderMap` 中是否有目标 `ContentProvider` 存在，有则返回，没有就会在注释 2 处调用 `IActivityManager` 的 `getContentProvider` 方法，最终会调用 AMS 的 `getContentProvider` 方法。注释 3 处的 `installProvider` 方法用来安装 `ContentProvider`，并将注释 2 处返回的 `ContentProvider` 相关的数据存储在 `mProviderMap` 中，起到缓存的作用，这样使用相同的 `Content Provider` 时，就不需要每次都要调用 AMS 的 `getContentProvider` 方法了。接着查看 AMS 的 `getContentProvider` 方法，代码如下所示：

frameworks/base/services/core/java/com/android/server/am/ActivityManagerService.java

```

@Override
public final ContentProviderHolder getContentProvider(
    IApplicationThread caller, String name, int userId, boolean stable) {
    enforceNotIsolatedCaller("getContentProvider");
    if (caller == null) {
        String msg = "null IApplicationThread when getting content provider "
            + name;
        Slog.w(TAG, msg);
        throw new SecurityException(msg);
    }
}

```

```

    }
    return getServiceProviderImpl(caller, name, null, stable, userId);
}

```

getServiceProvider 方法返回了 getServiceProviderImpl 方法：

frameworks/base/services/core/java/com/android/server/am/ActivityManagerService.java

```

private ContentProviderHolder getServiceProviderImpl(IApplicationThread caller,
    String name, IBinder token, boolean stable, int userId) {
    ...
    ProcessRecord proc = getProcessRecordLocked(cpi.processName,
cpr.appInfo.uid, false);//1
    if (proc != null && proc.thread != null && !proc.killed) {
        ...
        if (!proc.pubProviders.containsKey(cpi.name)) {
            checkTime(startTime, "getServiceProviderImpl: scheduling install");
            proc.pubProviders.put(cpi.name, cpr);
            try {
                proc.thread.scheduleInstallProvider(cpi);//2
            } catch (RemoteException e) {
            }
        }
    } else {
        checkTime(startTime, "getServiceProviderImpl: before start
        process");
        proc = startProcessLocked(cpi.processName, cpr.appInfo,
false, 0, "content provider", new
ComponentName(cpi.applicationInfo.packageName,
cpi.name), false, false, false);//3
        checkTime(startTime, "getServiceProviderImpl: after start
        process");
        ...
    }
    ...
}

```

getServiceProviderImpl 方法的代码很多，这里只截取了关键的部分。在注释 1 处通过 getProcessRecordLocked 方法来获取目标 ContentProvider 的应用程序进程信息，这些信息用 ProcessRecord 类型的 proc 来表示，如果该应用程序进程已经启动就会调用注释 2 处的代码，否则就会调用注释 3 处的 startProcessLocked 方法来启动进程。此前我们都假设应用程序进程已经启动的情况，这里假设 ContentProvider 的应用程序进程还没有启动，应用程序进程启动最终会调用 ActivityThread 的 main 方法，不了解的读者请查看本书第 2 章的内容。

ActivityThread 的 main 方法如下所示：

frameworks/base/core/java/android/app/ActivityThread.java

```
public static void main(String[] args) {
    ...
    Looper.prepareMainLooper();//1
    ActivityThread thread = new ActivityThread();//2
    thread.attach(false);
    if (sMainThreadHandler == null) {
        sMainThreadHandler = thread.getHandler();
    }
    if (false) {
        Looper.myLooper().setMessageLogging(new
            LogPrinter(Log.DEBUG, "ActivityThread"));
    }
    Trace.traceEnd(Trace.TRACE_TAG_ACTIVITY_MANAGER);
    Looper.loop();//3
    throw new RuntimeException("Main thread loop unexpectedly exited");
}
```

在注释 1 处通过 prepareMainLooper 方法在 ThreadLocal 中获取 Looper，并在注释 3 处开启消息循环。在注释 2 处创建了 ActivityThread，紧接着调用了它的 attach 方法：

frameworks/base/core/java/android/app/ActivityThread.java

```
private void attach(boolean system) {
    ....
    final IActivityManager mgr = ActivityManager.getService();//1
    try {
        mgr.attachApplication(mAppThread);//2
    } catch (RemoteException ex) {
        throw ex.rethrowFromSystemServer();
    }
    ...
}
```

注释 1 处得到 IActivityManager，在注释 2 处调用 IActivityManager 的 attachApplication 方法，并将 ApplicationThread 类型的 mAppThread 对象传进去，最终调用的是 AMS 的 attachApplication 方法。query 方法到 AMS 的调用过程（省略应用程序进程启动过程）的时序图如图 4-15 所示。

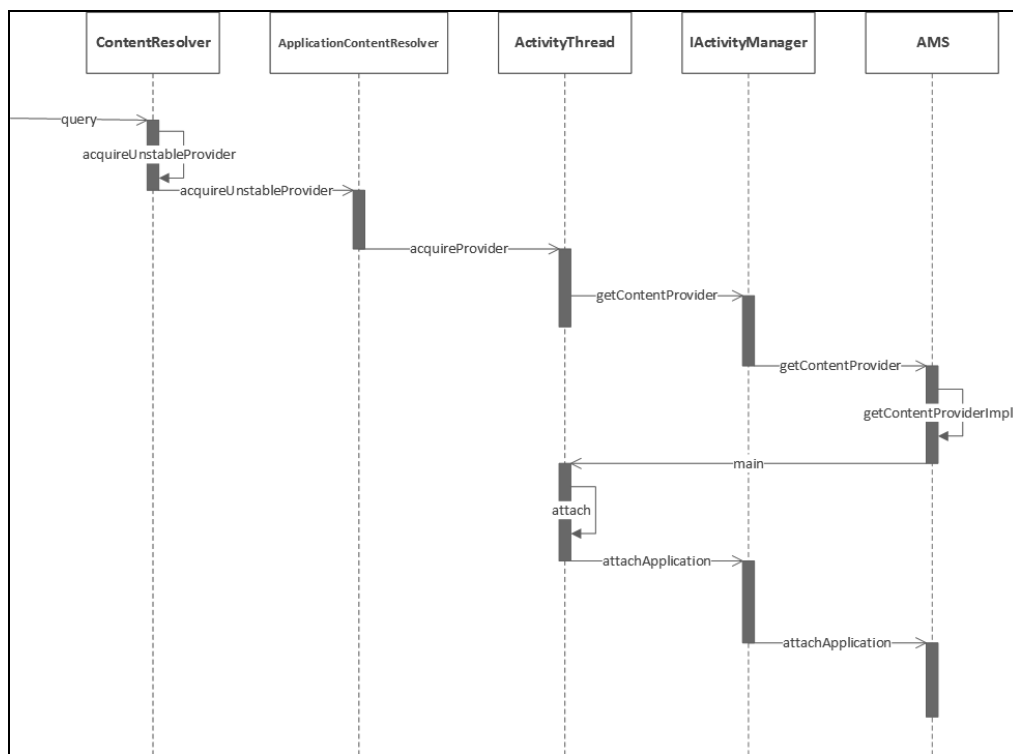


图 4-15 query 方法到 AMS 的调用过程的时序图

4.5.2 AMS启动Content Provider的过程

AMS 启动 Content Provider 的时序图如图 4-16 所示。

我们接着来查看 AMS 的 attachApplication 方法，如下所示：

```
frameworks/base/services/core/java/com/android/server/am/ActivityManagerService.java
```

```

@Override
public final void attachApplication(IApplicationThread thread) {
    synchronized (this) {
        int callingPid = Binder.getCallingPid();
        final long origId = Binder.clearCallingIdentity();
        attachApplicationLocked(thread, callingPid);
        Binder.restoreCallingIdentity(origId);
    }
}

```

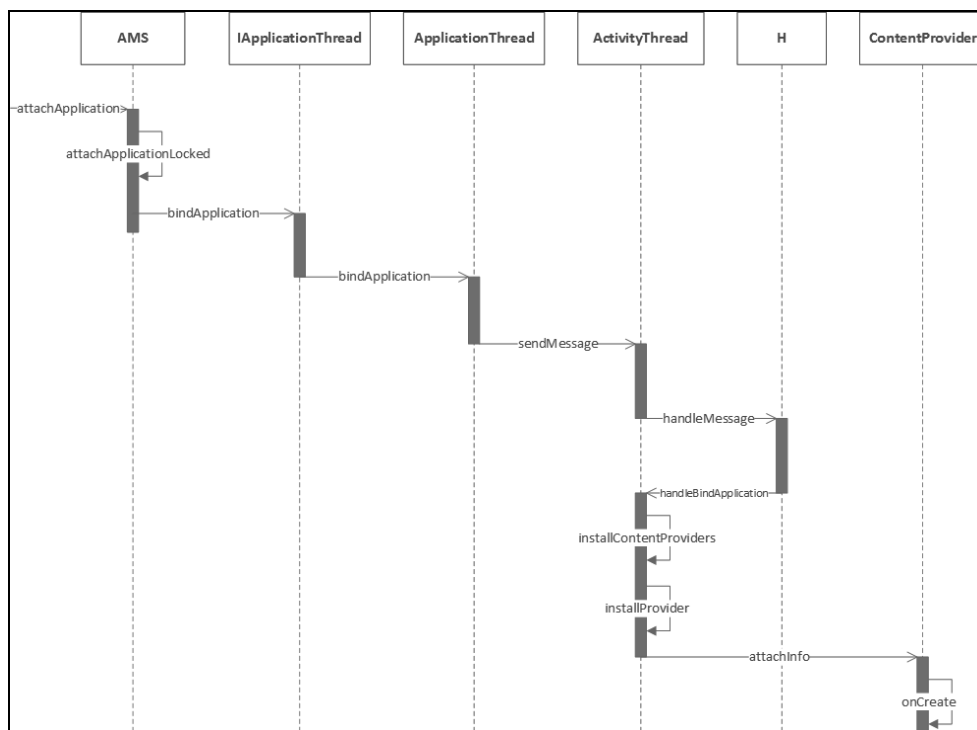


图 4-16 AMS 启动 Content Provider 的过程的时序图

在 attachApplication 方法中又调用了 attachApplicationLocked 方法：

frameworks/base/services/core/java/com/android/server/am/ActivityManagerService.java

```

private final boolean attachApplicationLocked(IApplicationThread thread,
    int pid) {
    ...
    thread.bindApplication(processName, appInfo, providers, app.instrumentationClass,
        profilerInfo, app.instrumentationArguments, app.instrumentationWatcher,
        app.instrumentationUiAutomationConnection, testMode,
        mBinderTransactionTrackingEnabled, enableTrackAllocation,
        isRestrictedBackupMode || !normalMode, app.persistent,
        new Configuration(mConfiguration), app.compat,
        getCommonServicesLocked(app.isolated),
        mCoreSettingsObserver.getCoreSettingsLocked());
    ...
}

```

在 attachApplicationLocked 方法中调用了 thread 的 bindApplication 方法，thread 是 IApplicationThread 类型的，这里和 IActivityManager 一样采用了 AIDL，实现 bindApplication

方法的不再是 Android 7.0 的 `ApplicationThreadProxy` 类，而是 `ApplicationThread` 类，它是 `ActivityThread` 的内部类，如下所示：

`frameworks/base/core/java/android/app/ActivityThread.java`

```
public final void bindApplication(String processName, ApplicationInfo appInfo,
    List<ProviderInfo> providers, ComponentName instrumentationName,
    ProfilerInfo profilerInfo, Bundle instrumentationArgs,
    IInstrumentationWatcher instrumentationWatcher,
    IUiAutomationConnection instrumentationUiConnection, int debugMode,
    boolean enableBinderTracking, boolean trackAllocation,
    boolean isRestrictedBackupMode, boolean persistent, Configuration config,
    CompatibilityInfo compatInfo, Map services, Bundle coreSettings,
    String buildSerial) {
    if (services != null) {
        ServiceManager.initServiceCache(services);
    }
    setCoreSettings(coreSettings);
    AppBindData data = new AppBindData();
    data.processName = processName;
    data.appInfo = appInfo;
    data.providers = providers;
    data.instrumentationName = instrumentationName;
    data.instrumentationArgs = instrumentationArgs;
    ...
    sendMessage(H.BIND_APPLICATION, data);
}
```

在 `bindApplication` 方法中最后调用 `sendMessage` 方法向 H 发送 `BIND_APPLICATION` 类型消息，H 的 `handleMessage` 方法如下所示：

`frameworks/base/core/java/android/app/ActivityThread.java`

```
public void handleMessage(Message msg) {
    if (DEBUG_MESSAGES) Slog.v(TAG, ">>> handling: " + codeToString(msg.what));
    switch (msg.what) {
        ...
        case BIND_APPLICATION:
            Trace.traceBegin(Trace.TRACE_TAG_ACTIVITY_MANAGER, "bindApplication");
            AppBindData data = (AppBindData)msg.obj;
            handleBindApplication(data);
            Trace.traceEnd(Trace.TRACE_TAG_ACTIVITY_MANAGER);
            break;
        ...
    }
    ...
}
```

我们接着查看 handleBindApplication 方法：

frameworks/base/core/java/android/app/ActivityThread.java

```
private void handleBindApplication(AppBindData data) {
    ...
    final ContextImpl appContext = ContextImpl.createAppContext(this, data.info);//1
    try {
        final ClassLoader cl = instrContext.getClassLoader();
        mInstrumentation = (Instrumentation)
            cl.loadClass(data.instrumentationName.getClassName()).
            newInstance();//2
    } catch (Exception e) {
        ...
    }
    final ComponentName component = new ComponentName(ii.packageName, ii.name);
    mInstrumentation.init(this, instrContext, appContext, component,
        data.instrumentationWatcher, data.instrumentationUiAutomation
        Connection);//3
    ...
    Application app = data.info.makeApplication(data.restrictedBackupMode,
        null);//4
    mInitialApplication = app;
    if (!data.restrictedBackupMode) {
        if (!ArrayUtils.isEmpty(data.providers)) {
            installContentProviders(app, data.providers);//5
            mH.sendMessageDelayed(H.ENABLE_JIT, 10*1000);
        }
    }
    ...
    mInstrumentation.callApplicationOnCreate(app);//6
    ...
}
```

handleBindApplication 方法的代码很长，这里截取了主要的部分。在注释 1 处创建了 ContextImpl。在注释 2 处通过反射创建 Instrumentation 并在注释 3 处初始化 Instrumentation。在注释 4 处创建 Application 并且在注释 6 处调用 Application 的 onCreate 方法，这意味着 Content Provider 所在的应用程序已经启动，在应用程序启动之前，在注释 5 处调用 installContentProviders 方法来启动 Content Provider，代码如下所示：

frameworks/base/core/java/android/app/ActivityThread.java

```
private void installContentProviders(
    Context context, List<ProviderInfo> providers) {
    final ArrayList<ContentProviderHolder> results = new ArrayList<>();
    for (ProviderInfo cpi : providers) {//1
```



```

if (DEBUG_PROVIDER) {
    StringBuilder buf = new StringBuilder(128);
    buf.append("Pub ");
    buf.append(cpi.authority);
    buf.append(": ");
    buf.append(cpi.name);
    Log.i(TAG, buf.toString());
}
ContentProviderHolder cph = installProvider(context, null, cpi,
false /*noisy*/, true /*noReleaseNeeded*/, true /*stable* */);//2
if (cph != null) {
    cph.noReleaseNeeded = true;
    results.add(cph);
}
}
try {
    ActivityManager.getService().publishContentProviders(
        getApplicationThread(), results);//3
} catch (RemoteException ex) {
    throw ex.rethrowFromSystemServer();
}
}

```

在注释 1 处遍历当前应用程序进程的 ProviderInfo 列表，得到每个 Content Provider 的 ProviderInfo（存储 Content Provider 的信息），并在注释 2 处调用 installProvider 方法来启动这些 Content Provider。在注释 3 处通过 AMS 的 publishContentProviders 方法将这些 Content Provider 存储在 AMS 的 mProviderMap 中，这个 mProviderMap 在前面提到过，起到缓存的作用，防止每次使用相同的 Content Provider 时都会调用 AMS 的 getContentProvider 方法。下面来查看 installProvider 方法是如何启动 Content Provider 的，installProvider 方法如下所示：

```

frameworks/base/core/java/android/app/ActivityThread.java

private IActivityManager.ContentProviderHolder installProvider(Context context,
    IActivityManager.ContentProviderHolder holder, ProviderInfo info,
    boolean noisy, boolean noReleaseNeeded, boolean stable) {
    ContentProvider localProvider = null;

    ...

    final java.lang.ClassLoader cl = c.getClassLoader();
    localProvider = (ContentProvider)cl.
        loadClass(info.name).newInstance();//1
    provider = localProvider.getIContentProvider();
    if (provider == null) {
        ...
        return null;
    }
}

```

```

    }
    if (DEBUG_PROVIDER) Slog.v(
        TAG, "Instantiating local provider " + info.name);
    localProvider.attachInfo(c, info);//2
} catch (java.lang.Exception e) {
    ...
}
return null;
}
}
...
return retHolder;
}

```

在注释 1 处通过反射来创建 `ContentProvider` 类型的 `localProvider` 对象，并在注释 2 处调用了它的 `attachInfo` 方法：

```
frameworks/base/core/java/android/content/ContentProvider.java
```

```

private void attachInfo(Context context, ProviderInfo info, boolean testing) {
    ...
    ContentProvider.this.onCreate();
}
}

```

在 `attachInfo` 方法中调用了 `onCreate` 方法，它是一个抽象方法，这样 `Content Provider` 就启动完毕。当然这只是 `Content Provider` 启动过程的一个分支，即应用程序进程没有启动的情况，还有一个分支是应用程序进程已经启动的情况，这就需要读者自行阅读源码了。

4.6 本章小结

本章的内容比较多，介绍了 Android 8.0 四大组件的启动过程，与 Android 7.0 主要区别是，与 AMS 进行进程间通信时采用的 AIDL 技术去掉了此前一直沿用的 `ActivityManagerProxy` 和 `ApplicationThreadProxy` 等代理类。四大组件的启动流程都大同小异，掌握了其中的一个，其他的也就很容易掌握，因为这些知识点都是触类旁通的。另外本章内容和第 3 章的内容有较大的关联，阅读本章能够更好地理解为什么要学习第 3 章应用程序进程启动这一知识点。另外本章经常用到上下文 `Context` 的知识点，这些知识点会在下一章进行介绍。

第 5 章

理解上下文 Context

关联章节：第 4 章 四大组件的工作过程

Context 也就是上下文对象，是 Android 常用的类，但是对于 Context，很多人都停留在会用的阶段，本章将带领大家从源码角度来分析 Context，从而更加深入地理解它。

Android 中的四大组件都会涉及 Context，因此我们在第 4 章经常会看到 Context 的身影，比如启动 Service 会调用 ContextWrapper 以及 ContextImpl 的 startService 方法，ContextWrapper 以及 ContextImpl 就是 Context 的关联类，理解这些 Context 的关联类可以更好地理解第 4 章的内容，本章内容和第 4 章的内容相辅相成，并且有一些重复的内容，对于重复的内容，本章不会赘述，建议先阅读第 4 章的内容再来阅读本章内容，阅读完本章后回过头重新阅读第 4 章的内容，这样你会有更多的发现。

5.1 Context 的关联类

Context 意为上下文，是一个应用程序环境信息的接口。

在开发中我们经常使用 Context，它的使用场景总的来说分为两大类，它们分别是：

- 使用 Context 调用方法，比如启动 Activity、访问资源、调用系统级服务等。
- 调用方法时传入 Context，比如弹出 Toast、创建 Dialog 等。

Activity、Service 和 Application 都间接地继承自 Context，因此我们可以计算出一个应用程序进程中有多少个 Context，这个数量等于 Activity 和 Service 的总个数加 1，1 指的是 Application 的数量。

Context 是一个抽象类，它的内部定义了很多方法以及静态常量，它的具体实现类为 ContextImpl。和 Context 相关联的类，除了 ContextImpl，还有 ContextWrapper、ContextThemeWrapper 和 Activity 等，如图 5-1 所示。

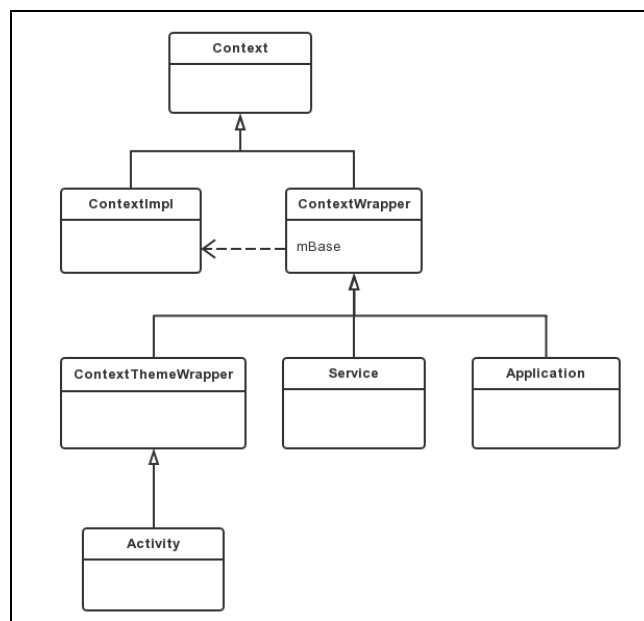


图 5-1 Context 的关联类

从图 5-1 中我们可以看出，ContextImpl 和 ContextWrapper 继承自 Context，ContextWrapper 内部包含 Context 类型的 mBase 对象，mBase 具体指向 ContextImpl。ContextImpl 提供了很多功能，但是外界需要使用并拓展 ContextImpl 的功能，因此设计上使用了装饰模式，ContextWrapper 是装饰类，它对 ContextImpl 进行包装，ContextWrapper 主要是起了方法传递的作用，ContextWrapper 中几乎所有的方法都是调用 ContextImpl 的相应方法来实现的。ContextThemeWrapper、Service 和 Application 都继承自 ContextWrapper，这样它们都可以通过 mBase 来使用 Context 的方法，同时它们也是装饰类，在 ContextWrapper 的基础上又添加了不同的功能。ContextThemeWrapper 中包含和主题相关的方法（比如 getTheme 方法），因此，需要主题的 Activity 继承 ContextThemeWrapper，而不需要主题的 Service 继承 ContextWrapper。

Context 的关联类采用了装饰模式，主要有以下的优点：

- 使用者（比如 Service）能够更方便地使用 Context。
- 如果 ContextImpl 发生了变化，它的装饰类 ContextWrapper 不需要做任何修改。
- ContextImpl 的实现不会暴露给使用者，使用者也不必关心 ContextImpl 的实现。
- 通过组合而非继承的方式，拓展 ContextImpl 的功能，在运行时选择不同的装饰类，实现不同的功能。

为了更好地理解 Context 的关联类的设计理念，就需要理解 Application、Activity、Service 的 Context 的创建过程，下面分别对它们进行介绍。

5.2 Application Context 的创建过程

我们通过调用 `getApplicationContext` 来获取应用程序全局的 Application Context，那么 Application Context 是如何创建的呢？在一个应用程序启动完成后，应用程序就会有一个全局的 Application Context，那么我们就从应用程序启动过程开始着手。Application Context 的创建过程的时序图如图 5-2 所示。

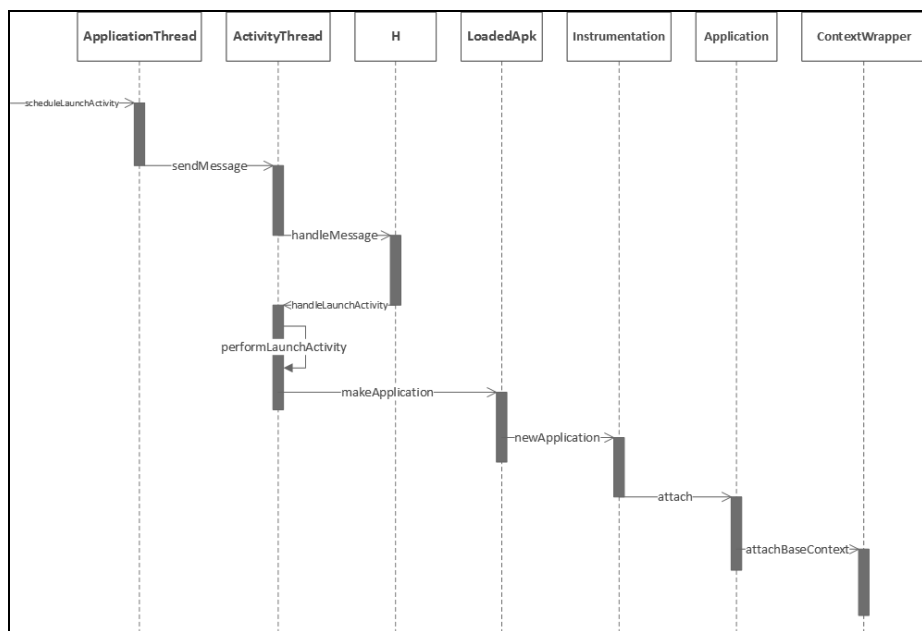


图 5-2 Application Context 的创建过程的时序图

ActivityThread 类作为应用程序进程的主线程管理类，它会调用它的内部类 ApplicationThread 的 scheduleLaunchActivity 方法来启动 Activity，如下所示：

```
frameworks/base/core/java/android/app/ActivityThread.java
```

```
private class ApplicationThread extends IApplicationThread.Stub {
    ...
    @Override
    public final void scheduleLaunchActivity(Intent intent, IBinder token, int
        ident, ActivityInfo info, Configuration curConfig, Configuration overrideConfig,
        CompatibilityInfo compatInfo, String referrer, IVoiceInteractor voiceInteractor,
        int procState, Bundle state, PersistableBundle persistentState,
        List<ResultInfo> pendingResults, List<ReferrerIntent> pendingNewIntents,
        boolean notResumed, boolean isForward, ProfilerInfo profilerInfo) {
        updateProcessState(procState, false);
        ActivityClientRecord r = new ActivityClientRecord();
        r.token = token;
        r.ident = ident;
        r.intent = intent;
        r.referrer = referrer;
        r.voiceInteractor = voiceInteractor;
        r.activityInfo = info;
        ...
        updatePendingConfiguration(curConfig);
        sendMessage(H.LAUNCH_ACTIVITY, r);
    }
    ...
}
```

在 ApplicationThread 的 scheduleLaunchActivity 方法中向 H 类发送 LAUNCH_ACTIVITY 类型的消息，目的是将启动 Activity 的逻辑放在主线程的消息队列中，这样启动 Activity 的逻辑会在主线程中执行。我们接着查看 H 类的 handleMessage 方法对 LAUNCH_ACTIVITY 类型的消息的处理：

```
frameworks/base/core/java/android/app/ActivityThread.java
```

```
private class H extends Handler {
    public static final int LAUNCH_ACTIVITY = 100;
    ...
    public void handleMessage(Message msg) {
        if (DEBUG_MESSAGES) Slog.v(TAG, ">>> handling: " + codeToString(msg.what));
        switch (msg.what) {
            case LAUNCH_ACTIVITY: {
                Trace.traceBegin(Trace.TRACE_TAG_ACTIVITY_MANAGER, "activityStart");
                final ActivityClientRecord r = (ActivityClientRecord) msg.obj;
```

```

        r.packageInfo = getPackageInfoNoCheck(
            r.activityInfo.applicationInfo, r.compatInfo); //1
        handleLaunchActivity(r, null, "LAUNCH_ACTIVITY"); //2
        Trace.traceEnd(Trace.TRACE_TAG_ACTIVITY_MANAGER);
    } break;

    ...
}

```

H 继承自 Handler, 是 ActivityThread 的内部类。在注释 1 处通过 getPackageInfoNoCheck 方法获得 LoadedApk 类型的对象, 并将该对象赋值给 ActivityClientRecord 的成员变量 packageInfo, 其中 LoadedApk 用来描述已加载的 APK 文件。在注释 2 处调用了 ActivityThread 的 handleLaunchActivity 方法, 如下所示:

frameworks/base/core/java/android/app/ActivityThread.java

```

private void handleLaunchActivity(ActivityClientRecord r, Intent customIntent,
    String reason) {
    ...
    Activity a = performLaunchActivity(r, customIntent);
    ...
}

```

在 handleLaunchActivity 方法中调用了 ActivityThread 的 performLaunchActivity 方法:

frameworks/base/core/java/android/app/ActivityThread.java

```

private Activity performLaunchActivity(ActivityClientRecord r, Intent customIntent) {
    ...
    try {
        Application app = r.packageInfo.makeApplication(false, mInstrumentation);
        ...
    }
    ...
    return activity;
}

```

在 performLaunchActivity 方法中有很多重要的逻辑, 这里只保留了和 Application Context 相关的逻辑, 想要了解更多 performLaunchActivity 方法中的逻辑请查看 4.1.3 节的内容。ActivityClientRecord 的成员变量 packageInfo 是 LoadedApk 类型的, 我们接着来查看 LoadedApk 的 makeApplication 方法, 如下所示:

frameworks/base/core/java/android/app/LoadedApk.java

```

public Application makeApplication(boolean forceDefaultAppClass,

```

```

        Instrumentation instrumentation) {
    if (mApplication != null) { //1
        return mApplication;
    }
    Trace.traceBegin(Trace.TRACE_TAG_ACTIVITY_MANAGER, "makeApplication");
    Application app = null;
    String appClass = mApplicationInfo.className;
    if (forceDefaultAppClass || (appClass == null)) {
        appClass = "android.app.Application";
    }
    try {
        java.lang.ClassLoader cl = getClassLoader();
        if (!mPackageName.equals("android")) {
            Trace.traceBegin(Trace.TRACE_TAG_ACTIVITY_MANAGER,
                "initializeJavaContextClassLoader");
            initializeJavaContextClassLoader();
            Trace.traceEnd(Trace.TRACE_TAG_ACTIVITY_MANAGER);
        }
        ContextImpl appContext = ContextImpl.createAppContext(mActivityThread,
            this); //2
        app = mActivityThread.mInstrumentation.newApplication(
            cl, appClass, appContext); //3
        appContext.setOuterContext(app); //4
    } catch (Exception e) {
        ...
    }
    mActivityThread.mAllApplications.add(app);
    mApplication = app; //5
    ...
    return app;
}

```

在注释 1 处如果 `mApplication` 不为 `null` 则返回 `mApplication`，这里假设是第一次启动应用程序，因此 `mApplication` 为 `null`。在注释 2 处通过 `ContextImpl` 的 `createAppContext` 方法来创建 `ContextImpl`。注释 3 处的代码用来创建 `Application`，在 `Instrumentation` 的 `newApplication` 方法中传入了 `ClassLoader` 类型的对象以及注释 2 处创建的 `ContextImpl`。在注释 4 处将 `Application` 赋值给 `ContextImpl` 的 `Context` 类型的成员变量 `mOuterContext`，这样 `ContextImpl` 中也包含了 `Application` 的引用。在注释 5 处将 `Application` 赋值给 `LoadedApk` 的成员变量 `mApplication`，这个 `mApplication` 是 `Application` 类型的对象，它用来代表 `Application Context`，在 `Application Context` 的获取过程中我们会再次提到 `mApplication`。下面来查看注释 3 处的 `Application` 是如何创建的，`Instrumentation` 的 `newApplication` 方法如下所示：


```
frameworks/base/core/java/android/app/Instrumentation.java
```

```
static public Application newApplication(Class<?> clazz, Context context)
throws InstantiationException, IllegalAccessException, ClassNotFoundException
{
    Application app = (Application)clazz.newInstance();
    app.attach(context); //1
    return app;
}
```

Instrumentation 中有两个 newApplication 重载方法，最终会调用上面这个重载方法。注释 1 处通过反射来创建 Application，并调用了 Application 的 attach 方法，将 ContextImpl 传进去，最后返回该 Application，Application 的 attach 方法如下所示：

```
frameworks/base/core/java/android/app/Application.java
```

```
/* package */ final void attach(Context context) {
    attachBaseContext(context);
    mLoadedApk = ContextImpl.getImpl(context).mPackageInfo;
}
```

在 attach 方法中调用了 attachBaseContext 方法，它在 Application 的父类 ContextWrapper 中实现，代码如下所示：

```
frameworks/base/core/java/android/content/ContextWrapper.java
```

```
protected void attachBaseContext(Context base) {
    if (mBase != null) {
        throw new IllegalStateException("Base context already set");
    }
    mBase = base;
}
```

这个 base 一路传递过来指的是 ContextImpl，它是 Context 的实现类，将 ContextImpl 赋值给 ContextWrapper 的 Context 类型的成员变量 mBase，这样在 ContextWrapper 中就可以使用 Context 的方法，而 Application 继承自 ContextWrapper，同样可以使用 Context 的方法。Application 的 attach 方法的作用就是使 Application 可以使用 Context 的方法，这样 Application 才可以用来代表 Application Context。

Application Context 的创建过程就讲到这里，接下来我们来学习 Application Context 的获取过程。

5.3 Application Context的获取过程

当我们熟知了 Application Context 的创建过程之后，那么它的获取过程会非常好理解。我们通过调用 `getApplicationContext` 方法来获得 Application Context，`getApplicationContext` 方法在 `ContextWrapper` 中实现，如下所示：

```
frameworks/base/core/java/android/content/ContextWrapper.java
```

```
@Override
public Context getApplicationContext() {
    return mBase.getApplicationContext();
}
```

`mBase` 指的是 `ContextImpl`，我们来查看 `ContextImpl` 的 `getApplicationContext` 方法：

```
frameworks/base/core/java/android/app/ContextImpl.java
```

```
@Override
public Context getApplicationContext() {
    return (mPackageInfo != null) ?
        mPackageInfo.getApplication() : mMainThread.getApplication();
}
```

如果 `LoadedApk` 类型的 `mPackageInfo` 不为 `null`，则调用 `LoadedApk` 的 `getApplication` 方法，否则调用 `ActivityThread` 的 `getApplication` 方法。由于应用程序这时已经启动，因此 `LoadedApk` 不会为 `null`，则会调用 `LoadedApk` 的 `getApplication` 方法，如下所示：

```
frameworks/base/core/java/android/app/LoadedApk.java
```

```
Application getApplication() {
    return mApplication;
}
```

这里的 `mApplication` 我们应该很熟悉，它在上文 `LoadedApk` 的 `makeApplication` 方法的注释 5 处被赋值。这样我们通过 `getApplicationContext` 方法就获取到了 Application Context。

5.4 Activity的Context创建过程

想要在 Activity 中使用 Context 提供的方法，务必要先创建 Context。Activity 的 Context 会在 Activity 的启动过程中被创建，在 4.1.3 节中讲到了 `ActivityThread` 启动 Activity 的过程，我们就从这里开始分析。Activity 的 Context 创建过程的时序图如图 5-3 所示。

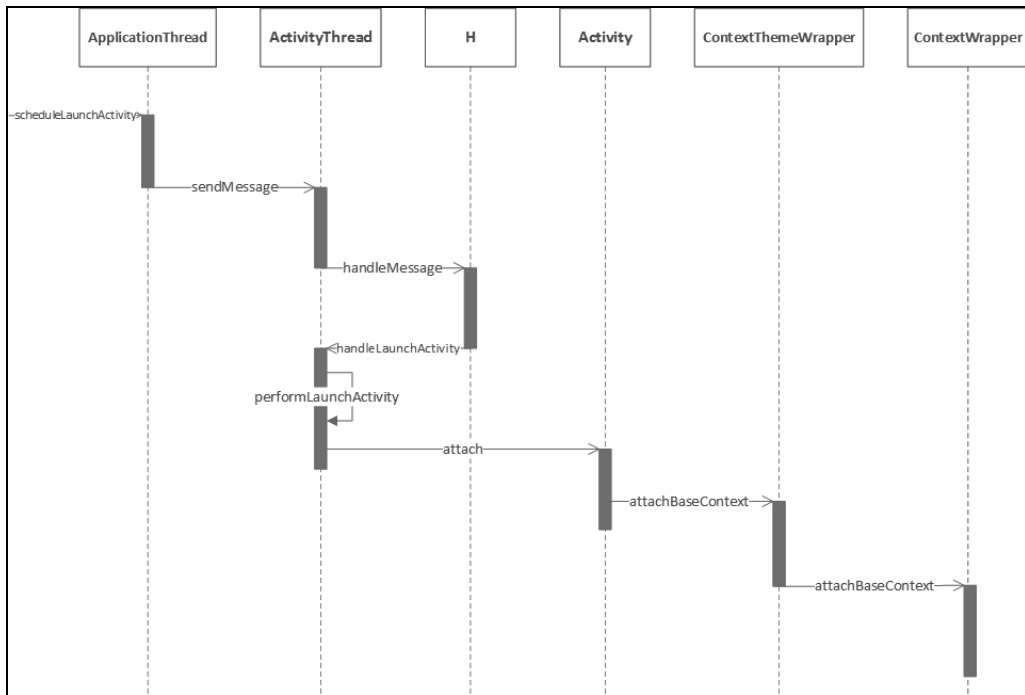


图 5-3 Activity 的 Context 创建过程的时序图

ActivityThread 是应用程序进程的主线程管理类，它的内部类 ApplicationThread 会调用 scheduleLaunchActivity 方法来启动 Activity，scheduleLaunchActivity 方法如下所示：

frameworks/base/core/java/android/app/ActivityThread.java

```

@Override
public final void scheduleLaunchActivity(Intent intent, IBinder token, int
    ident, ActivityInfo info, Configuration curConfig, Configuration overrideConfig,
    CompatibilityInfo compatInfo, String referrer, IVoiceInteractor voiceInteractor,
    int procState, Bundle state, PersistableBundle persistentState,
    List<ResultInfo> pendingResults, List<ReferrerIntent> pendingNewIntents,
    boolean notResumed, boolean isForward, ProfilerInfo profilerInfo) {
    updateProcessState(procState, false);
    ActivityClientRecord r = new ActivityClientRecord();
    r.token = token;
    r.ident = ident;
    r.intent = intent;
    r.referrer = referrer;
    ...
    sendMessage(H.LAUNCH_ACTIVITY, r);
}
  
```

`scheduleLaunchActivity` 方法将启动 Activity 的参数封装成 `ActivityClientRecord`, `sendMessage` 方法向 H 类发送类型为 `LAUNCH_ACTIVITY` 的消息, 并将 `ActivityClientRecord` 传递过去。`sendMessage` 方法的目的是将启动 Activity 的逻辑放在主线程的消息队列中, 这样启动 Activity 的逻辑就会在主线程中执行。H 类的 `handleMessage` 方法会对 `LAUNCH_ACTIVITY` 类型的消息进行处理, 其中调用了 `ActivityThread` 的 `handleLaunchActivity` 方法, 而在 `handleLaunchActivity` 方法中又调用了 `ActivityThread` 的 `performLaunchActivity` 方法, 这一过程在 5.2 节已经讲过了, 我们直接来查看 `ActivityThread` 的 `performLaunchActivity` 方法:

```
frameworks/base/core/java/android/app/ActivityThread.java
```

```
private Activity performLaunchActivity(ActivityClientRecord r, Intent customIntent) {
    ...
    ContextImpl appContext = createBaseContextForActivity(r); //1
    Activity activity = null;
    try {
        java.lang.ClassLoader cl = appContext.getClassLoader();
        activity = mInstrumentation.newActivity(
            cl, component.getClassName(), r.intent); //2
        StrictMode.incrementExpectedActivityCount(activity.getClass());
        r.intent.setExtrasClassLoader(cl);
        r.intent.prepareToEnterProcess();
        if (r.state != null) {
            r.state.setClassLoader(cl);
        }
    } catch (Exception e) {
        ...
    }
    try {
        ...
        if (activity != null) {
            ...
            appContext.setOuterContext(activity); //3
            /**
             *4
             */
            activity.attach(appContext, this, getInstrumentation(), r.token, r.ident,
                app, r.intent, r.activityInfo, title, r.parent, r.embeddedID,
                r.lastNonConfigurationInstances, config, r.referrer, r.voiceInteractor,
                window, r.configCallback);
            ...
            if (r.isPersistable()) {
                mInstrumentation.callActivityOnCreate(activity, r.state, r.persistent
                    State); //5
            }
        }
    }
}
```

```

        } else {
            mInstrumentation.callActivityOnCreate(activity, r.state);
        }
        ...
    }
    r.paused = true;
    mActivities.put(r.token, r);
} catch (SuperNotCalledException e) {
    ...
}
return activity;
}

```

在 `performLaunchActivity` 方法中有很多重要的逻辑，这里只保留了 Activity 的 Context 相关的逻辑。在注释 2 处用来创建 Activity 的实例。在注释 1 处通过 `createBaseContextForActivity` 方法来创建 Activity 的 `ContextImpl`，并将 `ContextImpl` 传入注释 4 处的 activity 的 `attach` 方法中。在注释 3 处调用了 `ContextImpl` 的 `setOuterContext` 方法，将此前创建的 Activity 实例赋值给 `ContextImpl` 的成员变量 `mOuterContext`，这样 `ContextImpl` 也可以访问 Activity 的变量和方法。在注释 5 处 `m.Instrumentation` 的 `callActivityOnCreate` 方法中会调用 Activity 的 `onCreate` 方法。我们查看注释 1 处的 `createBaseContextForActivity` 方法：

frameworks/base/core/java/android/app/ActivityThread.java

```

private ContextImpl createBaseContextForActivity(ActivityClientRecord r) {
    ...
    ContextImpl appContext = ContextImpl.createActivityContext(
        this, r.packageInfo, r.activityInfo, r.token, displayId, r.overrideConfig);
    ...
    return appContext;
}

```

在 `createBaseContextForActivity` 方法中会调用 `ContextImpl` 的 `createActivityContext` 方法来创建 `ContextImpl`。我们回到 `ActivityThread` 的 `performLaunchActivity` 方法，查看注释 4 处的 Activity 的 `attach` 方法，如下所示：

frameworks/base/core/java/android/app/Activity.java

```

final void attach(Context context, ActivityThread aThread,
    Instrumentation instr, IBinder token, int ident,
    Application application, Intent intent, ActivityInfo info,
    CharSequence title, Activity parent, String id,
    NonConfigurationInstances lastNonConfigurationInstances,
    Configuration config, String referrer, IVoiceInteractor voiceInteractor,

```

```

        Window window, ActivityConfigCallback activityConfigCallback)
    {
        attachBaseContext(context); //1
        mFragments.attachHost(null /*parent*/);
        mWindow = new PhoneWindow(this, window, activityConfigCallback); //2
        mWindow.setWindowControllerCallback(this);
        mWindow.setCallback(this); //3
        mWindow.setOnWindowDismissedCallback(this);
        mWindow.getLayoutInflater().setPrivateFactory(this);
        ...
        mWindow.setWindowManager((WindowManager)context.getSystemService(
            Context.WINDOW_SERVICE), mToken, mComponent.flattenToString(),
            (info.flags & ActivityInfo.FLAG_HARDWARE_ACCELERATED) != 0); //4
        if (mParent != null) {
            mWindow.setContainer(mParent.getWindow());
        }
        mWindowManager = mWindow.getWindowManager(); //5
        mCurrentConfig = config;
        mWindow.setColorMode(info.colorMode);
    }

```

在注释 2 处创建 PhoneWindow，它代表应用程序窗口。PhoneWindow 在运行中会间接触发很多事件，比如点击、菜单弹出、屏幕焦点变化等事件，这些事件需要转发给与 PhoneWindow 关联的 Activity，转发操作通过 Window.Callback 接口实现，Activity 实现了这个接口。在注释 3 处将当前 Activity 通过 Window 的 setCallback 方法传递给 PhoneWindow。在注释 4 处为 PhoneWindow 设置 WindowManager，在注释 5 处获取 WindowManager 并赋值给 Activity 的成员变量 mWindowManager，这样在 Activity 中就可以通过 getWindowManager 方法来获取 WindowManager。注释 1 处的 attachBaseContext 方法在 ContextThemeWrapper 中实现，如下所示：

```
frameworks/base/core/java/android/view/ContextThemeWrapper.java
```

```

@Override
protected void attachBaseContext(Context newBase) {
    super.attachBaseContext(newBase);
}

```

attachBaseContext 方法接着调用 ContextThemeWrapper 的父类 ContextWrapper 的 attachBaseContext 方法：

```
frameworks/base/core/java/android/content/ContextWrapper.java
```

```
protected void attachBaseContext(Context base) {
```

```

        if (mBase != null) {
            throw new IllegalStateException("Base context already set");
        }
        mBase = base;//1
    }

```

注释 1 处的 base 指的是一路传递过来的 Activity 的 ContextImpl，将它赋值给 ContextWrapper 的成员变量 mBase。这样 ContextWrapper 的功能就可以交由 ContextImpl 来处理，举个例子，如下所示：

frameworks/base/core/java/android/content/ContextWrapper.java

```

Override
public Resources.Theme getTheme() {
    return mBase.getTheme();
}

```

当我们调用 ContextWrapper 的 getTheme 方法时，其实就是调用了 ContextImpl 的 getTheme 方法。Activity 的 Context 创建过程就讲到这里。总结一下，在启动 Activity 的过程中创建 ContextImpl，并赋值给 ContextWrapper 的成员变量 mBase。Activity 继承自 ContextWrapper 的子类 ContextThemeWrapper，这样在 Activity 中就可以使用 Context 中定义的方法了。

5.5 Service 的 Context 创建过程

Service 的 Context 创建过程与 Activity 的 Context 创建过程类似，是在 Service 的启动过程中被创建的。Service 的 Context 创建过程的时序图可以参考图 5-3，这里就不再给出。在 4.2.2 节中讲到了 ActivityThread 启动 Service 的过程，我们从这里开始分析。ActivityThread 的内部类 ApplicationThread 会调用 scheduleCreateService 方法来启动 Service，如下所示：

frameworks/base/core/java/android/app/ActivityThread.java

```

public final void scheduleCreateService(IBinder token,
    ServiceInfo info, CompatibilityInfo compatInfo, int processState) {
    updateProcessState(processState, false);
    CreateServiceData s = new CreateServiceData();
    s.token = token;
    s.info = info;
    s.compatInfo = compatInfo;
    sendMessage(H.CREATE_SERVICE, s);
}

```

sendMessage 方法向 H 类发送 CREATE_SERVICE 类型的消息，H 类的 handleMessage 方法会对 CREATE_SERVICE 类型的消息进行处理，其中调用了 ActivityThread 的 handleCreateService 方法：

frameworks/base/core/java/android/app/ActivityThread.java

```
private void handleCreateService(CreateServiceData data) {
    ...
    try {
        if (localLOGV) Slog.v(TAG, "Creating service " + data.info.name);
        ContextImpl context = ContextImpl.createAppContext(this, packageInfo);//1
        context.setOuterContext(service);
        Application app = packageInfo.makeApplication(false, mInstrumentation);
        service.attach(context, this, data.info.name, data.token, app,
            ActivityManager.getService());//2
        service.onCreate();
        mServices.put(data.token, service);
        try {
            ActivityManager.getService().serviceDoneExecuting(
                data.token, SERVICE_DONE_EXECUTING_ANON, 0, 0);
        } catch (RemoteException e) {
            throw e.rethrowFromSystemServer();
        }
    } catch (Exception e) {
        ...
    }
}
```

在注释 1 处通过 ContextImpl 的 createAppContext 方法创建了 ContextImpl，并将该 ContextImpl 传入注释 2 处 service 的 attach 方法中：

frameworks/base/core/java/android/app/Service.java

```
public final void attach(
    Context context,
    ActivityThread thread, String className, IBinder token,
    Application application, Object activityManager) {
    attachBaseContext(context);//1
    mThread = thread;
    mClassName = className;
    mToken = token;
    mApplication = application;
    mActivityManager = (IActivityManager)activityManager;
    mStartCompatibility = getApplicationInfo().targetSdkVersion
        < Build.VERSION_CODES.ECLAIR;
}
```


在注释 1 处调用了 ContextWrapper 的 attachBaseContext 方法，如下所示：

frameworks/base/core/java/android/content/ContextWrapper.java

```
protected void attachBaseContext(Context base) {  
    if (mBase != null) {  
        throw new IllegalStateException("Base context already set");  
    }  
    mBase = base;//1  
}
```

注释 1 处的 base 一路传递过来的是 ContextImpl，将 ContextImpl 赋值给 ContextWrapper 的 Context 类型的成员变量 mBase，这样在 ContextWrapper 中就可以使用 Context 的方法，而 Service 继承自 ContextWrapper，同样可以使用 Context 的方法。

5.6 本章小结

本章首先讲到了 Context 的关联类，又讲解了 Application、Activity 和 Service 的 Context 创建的过程，结合这些创建过程可以更好地理解 Context 的关联类的设计理念。同时本章的内容也会帮助读者更好地理解第 4 章的内容，建议阅读本章内容后，回过头阅读第 4 章的内容。

第 6 章

理解 ActivityManagerService

关联章节：第 2 章 Android 系统启动；第 3 章 应用程序进程启动过程；
第 4 章 四大组件的工作过程

在前面几章中介绍的 Android 系统启动过程、应用程序进程启动过程以及四大组件工作过程，都提及了 AMS，但都没有系统地来讲解它，本章就以 AMS 为主来进行讲解，其中有一些知识点与前面几章的知识点会有所重合，这里会尽量做到详尽讲解。阅读此章前，最好先阅读前面 5 章的内容。

6.1 AMS家族

AMS 处理的逻辑多而复杂，因此 AMS 并不是“孤军奋战”，而是有一些类和它“共同奋战”，这些类会帮助 AMS 完成相关逻辑，AMS 和这些“共同奋战”的类就称为 AMS 家族。Android 7.0 和 Android 8.0 对于 AMS 相关部分处理有较大的区别，为了更好地理解 AMS 家族，这里将分别介绍 Android 7.0 和 Android 8.0 的 AMS 家族。

6.1.1 Android 7.0 的AMS家族

ActivityManager 是一个和 AMS 相关联的类，它主要对运行中的 Activity 进行管理，这些管理工作并不是由 ActivityManager 来处理的，而是交由 AMS 来处理的。ActivityManager

中的方法会通过 ActivityManagerNative (以后简称 AMN) 的 getDefault 方法来得到 ActivityManagerProxy (以后简称 AMP), 通过 AMP 就可以和 AMN 进行通信, 而 AMN 是一个抽象类, 它将功能交由它的子类 AMS 来处理, 因此, AMP 就是 AMS 的代理类。AMS 作为系统服务, 很多 API 是不会暴露给 ActivityManager 的, 因此 ActivityManager 并不算是 AMS 家族的一份子。为了讲解 AMS 家族, 这里以 Android 7.0 的 Activity 启动过程来举例, 在 Activity 的启动过程中会调用 Instrumentation 的 execStartActivity 方法, 如下所示:

frameworks/base/core/java/android/app/Instrumentation.java

```
public ActivityResult execStartActivity(
    Context who, IBinder contextThread, IBinder token, Activity target,
    Intent intent, int requestCode, Bundle options) {
    ...
    try {
        intent.migrateExtraStreamToClipData();
        intent.prepareToLeaveProcess(who);
        int result = ActivityManagerNative.getDefault()
            .startActivity(whoThread, who.getBasePackageName(), intent,
                intent.resolveTypeIfNeeded(who.getContentResolver()),
                token, target != null ? target.mEmbeddedID : null,
                requestCode, 0, null, options);
        checkStartActivityResult(result, intent);
    } catch (RemoteException e) {
        throw new RuntimeException("Failure from system", e);
    }
    return null;
}
```

在 execStartActivity 方法中会调用 AMN 的 getDefault 来获取 AMS 的代理类 AMP。接着调用了 AMP 的 startActivity 方法, 先来查看 AMN 的 getDefault 方法做了什么, 如下所示:

frameworks/base/core/java/android/app/ActivityManagerNative.java

```
static public IActivityManager getDefault() {
    return gDefault.get();
}
private static final Singleton<IActivityManager> gDefault = new Singleton
<IActivityManager>() {
    protected IActivityManager create() {
        IBinder b = ServiceManager.getService("activity");//1
        if (false) {
            Log.v("ActivityManager", "default service binder = " + b);
        }
        IActivityManager am = asInterface(b);//2
    }
}
```

```

        if (false) {
            Log.v("ActivityManager", "default service = " + am);
        }
        return am;
    }
};
}

```

在 `getDefault` 方法中调用了 `gDefault` 的 `get` 方法, 我们接着往下看, `gDefault` 是一个 Singleton 类。在注释 1 处得到名为 “activity” 的 Service 引用, 也就是 `IBinder` 类型的 AMS 的引用。接着在注释 2 处将它封装成 AMP 类型对象, 并将它保存到 `gDefault` 中, 此后调用 AMN 的 `getDefault` 方法就会直接获得 AMS 的代理对象 AMP。注释 2 处的 `asInterface` 方法如下所示:

```

frameworks/base/core/java/android/app/ActivityManagerNative.java

static public IActivityManager asInterface(IBinder obj) {
    if (obj == null) {
        return null;
    }
    IActivityManager in =
        (IActivityManager)obj.queryLocalInterface(descriptor); //1
    if (in != null) {
        return in;
    }
    return new ActivityManagerProxy(obj); //2
}

```

注释 1 处的 `descriptor` 值为 `android.app.IActivityManager`, 注释 1 处的代码主要用来查询本地进程是否有 `IActivityManager` 接口的实现, 如果有则返回, 如果没有就在注释 2 处将 `IBinder` 类型的 AMS 引用封装成 AMP, AMP 的构造方法如下所示:

```

frameworks/base/core/java/android/app/ActivityManagerNative.java

class ActivityManagerProxy implements IActivityManager
{
    public ActivityManagerProxy(IBinder remote)
    {
        mRemote = remote;
    }
    ...
}

```

在 AMP 的构造方法中将 AMS 的引用赋值给变量 `mRemote`, 这样在 AMP 中就可以使用 AMS 了。其中 `IActivityManager` 是一个接口, AMN 和 AMP 都实现了这个接口, 用于实

现代理模式和 Binder 通信。再回到 Instrumentation 的 `execStartActivity` 方法，来查看 AMP 的 `startActivity` 方法，AMP 是 AMN 的内部类，代码如下所示：

```
frameworks/base/core/java/android/app/ActivityManagerNative.java
```

```
public int startActivity(IApplicationThread caller, String callingPackage, Intent
intent, String resolvedType, IBinder resultTo, String resultWho, int requestCode,
int startFlags, ProfilerInfo profilerInfo, Bundle options) throws RemoteException {
    ...
    data.writeInt(requestCode);
    data.writeInt(startFlags);
    ...
    mRemote.transact(START_ACTIVITY_TRANSACTION, data, reply, 0); //1
    reply.readException();+
    int result = reply.readInt();
    reply.recycle();
    data.recycle();
    return result;
}
```

首先将传入的参数写入到 Parcel 类型的 data 中。在注释 1 处，通过 IBinder 类型对象 mRemote (AMS 的引用) 向服务器端的 AMS 发送一个 START_ACTIVITY_TRANSACTION 类型的进程间通信请求。那么服务器端 AMS 就会从 Binder 线程池中读取客户端发来的数据，最终会调用 AMN 的 onTransact 方法，如下所示：

```
frameworks/base/core/java/android/app/ActivityManagerNative.java
```

```
@(进阶之神)Override
public boolean onTransact(int code, Parcel data, Parcel reply, int flags)
    throws RemoteException {
    switch (code) {
    case START_ACTIVITY_TRANSACTION:
    {
        ...
        int result = startActivity(app, callingPackage, intent, resolvedType,
            resultTo, resultWho, requestCode, startFlags, profilerInfo, options);
        reply.writeNoException();
        reply.writeInt(result);
        return true;
    }
    }
}
```

在 onTransact 方法中会调用 AMS 的 startActivity 方法，如下所示：

```
frameworks/base/services/core/java/com/android/server/am/ActivityManagerService.java
```

```
@Override
public final int startActivity(IApplicationThread caller, String callingPackage,
    Intent intent, String resolvedType, IBinder resultTo, String resultWho, int
    requestCode, int startFlags, ProfilerInfo profilerInfo, Bundle bOptions)
{
    return startActivityAsUser(caller, callingPackage, intent, resolvedType,
        resultTo, resultWho, requestCode, startFlags, profilerInfo, bOptions,
        UserHandle.getCallingUserId());
}
```

startActivity 方法最后会返回 startActivityAsUser 方法，如下所示：

```
frameworks/base/services/core/java/com/android/server/am/ActivityManagerService.java
```

```
@Override
public final int startActivityAsUser(IApplicationThread caller, String callingPackage,
    Intent intent, String resolvedType, IBinder resultTo, String resultWho, int
    requestCode, int startFlags, ProfilerInfo profilerInfo, Bundle bOptions, int
    userId) {
    enforceNotIsolatedCaller("startActivity");
    userId = mUserController.handleIncomingUser(Binder.getCallingPid(), Binder.
        getCallingUid(), userId, false, ALLOW_FULL_ONLY, "startActivity", null);
    return mActivityStarter.startActivityMayWait(caller, -1, callingPackage,
        intent, resolvedType, null, null, resultTo, resultWho, requestCode,
        startFlags, profilerInfo, null, null, bOptions, false, userId, null, null);
}
```

startActivityAsUser 方法最后会返回 ActivityStarter 的 startActivityMayWait 方法，这一调用过程已经脱离了本节要讲的 AMS 家族的范畴，因此这里不做介绍了，具体的调用过程可以查看 4.1.2 节的内容。在 Activity 的启动过程中提到了 AMP、AMN 和 AMS，它们共同组成了 AMS 家族的主要部分，如图 6-1 所示。

AMP 是 AMN 的内部类，它们都实现了 IActivityManager 接口，这样它们就可以实现代理模式，具体来讲是远程代理：AMP 和 AMN 是运行在两个进程中的，AMP 是 Client 端，AMN 则是 Server 端，而 Server 端中具体的功能都是由 AMN 的子类 AMS 来实现的，因此，AMP 就是 AMS 在 Client 端的代理类。AMN 又实现了 Binder 类，这样 AMP 和 AMS 就可以通过 Binder 来进行进程间通信。ActivityManager 通过 AMN 的 getDefault 方法得到 AMP，通过 AMP 就可以和 AMS 进行通信。除 ActivityManager 以外，有些想要与 AMS 进行通信的类也需要通过 AMP，如图 6-2 所示。

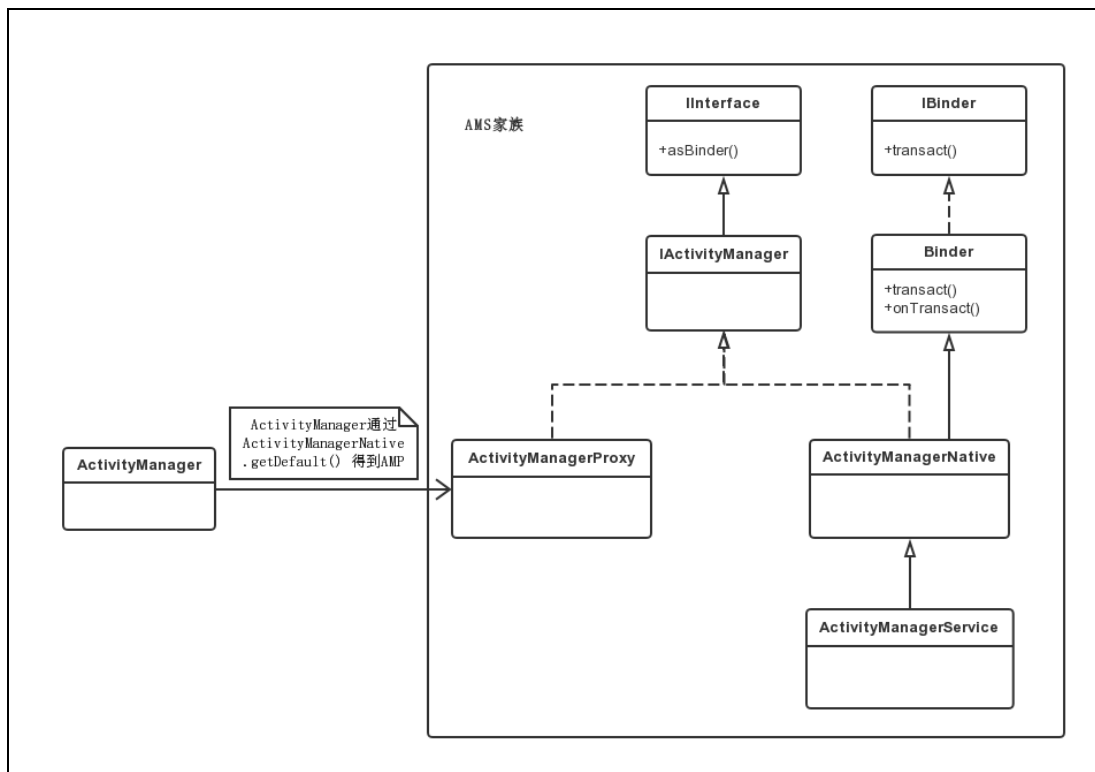


图 6-1 Android 7.0 AMS 家族

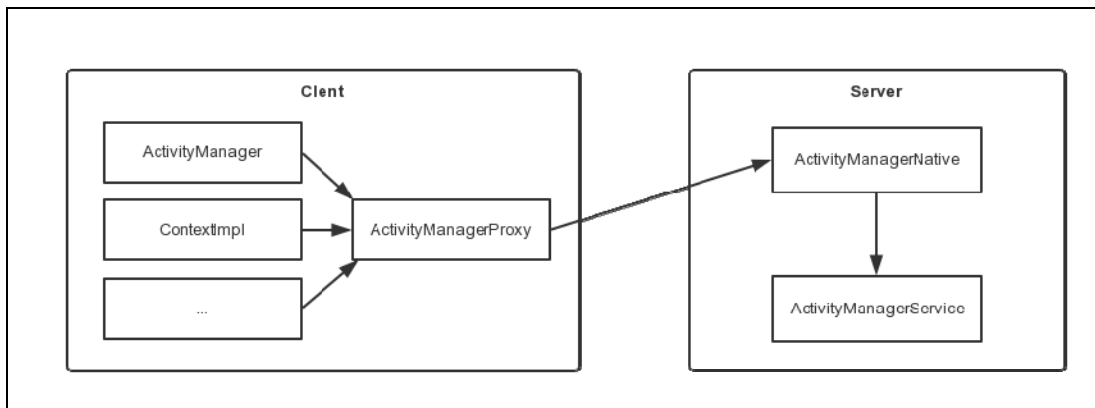


图 6-2 AMP 和 AMS 进行通信

从图 6-2 可以看出，除 ActivityManager 外，在第 5 章介绍的 ContextImpl 如果想要和 AMS 进行通信也需要先经过 AMP。

6.1.2 Android 8.0 的AMS家族

Android 8.0 的 AMS 家族与 Android 7.0 有一些差别，为了更好地理解这些差别，我们仍旧以 Activity 启动过程来举例，只不过版本是 Android 8.0，在 Activity 的启动过程中会调用 Instrumentation 的 `execStartActivity` 方法，如下所示：

frameworks/base/core/java/android/app/Instrumentation.java

```
public ActivityResult execStartActivity(
    Context who, IBinder contextThread, IBinder token, Activity target,
    Intent intent, int requestCode, Bundle options) {
    ...
    try {
        intent.migrateExtraStreamToClipData();
        intent.prepareToLeaveProcess(who);
        int result = ActivityManager.getService().startActivity(
            whoThread, who.getBasePackageName(), intent,
            intent.resolveTypeIfNeeded(who.getContentResolver()),
            token, target != null ? target.mEmbeddedID : null, requestCode, 0, null,
            options);
        checkStartActivityResult(result, intent);
    } catch (RemoteException e) {
        throw new RuntimeException("Failure from system", e);
    }
    return null;
}
```

ActivityManager 的 `getService` 方法，如下所示：

frameworks/base/core/java/android/app/ActivityManager.java

```
public static IActivityManager getService() {
    return IActivityManagerSingleton.get();
}

private static final Singleton<IActivityManager> IActivityManagerSingleton =
    new Singleton<IActivityManager>() {
        @Override
        protected IActivityManager create() {
            final IBinder b = ServiceManager.getService(Context.ACTIVITY_
                SERVICE); //1
            final IActivityManager am = IActivityManager.Stub.asInterface(b); //2
            return am;
        }
    };
```


getService 方法调用了 IActivityManagerSingleton 的 get 方法, IActivityManagerSingleton 是一个 Singleton 类。在注释 1 处得到名为“activity”的 Service 引用 (Context.ACTIVITY_SERVICE 的值为“activity”), 也就是 IBinder 类型的 AMS 的引用。接着在注释 2 处将它转换成 IActivityManager 类型的对象, 这段代码采用的是 AIDL, IActivityManager.java 类是由 AIDL 工具在编译时自动生成的, IActivityManager.aidl 的文件路径为 frameworks/base/core/java/android/app/IActivityManager.aidl。要实现进程间通信, 服务器端也就是 AMS 只需要继承 IActivityManager.Stub 类并实现相应的方法就可以了。采用 AIDL 后就不需要使用 AMS 的代理类 AMP 了, 因此 Android 8.0 去掉了 AMP, 代替它的是 IActivityManager, 它是 AMS 在本地的代理。我们回到 Instrumentation 的 execStartActivity 方法, 在注释 1 处实际上调用的是 AMS 的 execStartActivity 方法。剩下的调用过程就不再介绍了, 我们来查看 Android 8.0 的 AMS 家族, 如图 6-3 所示。

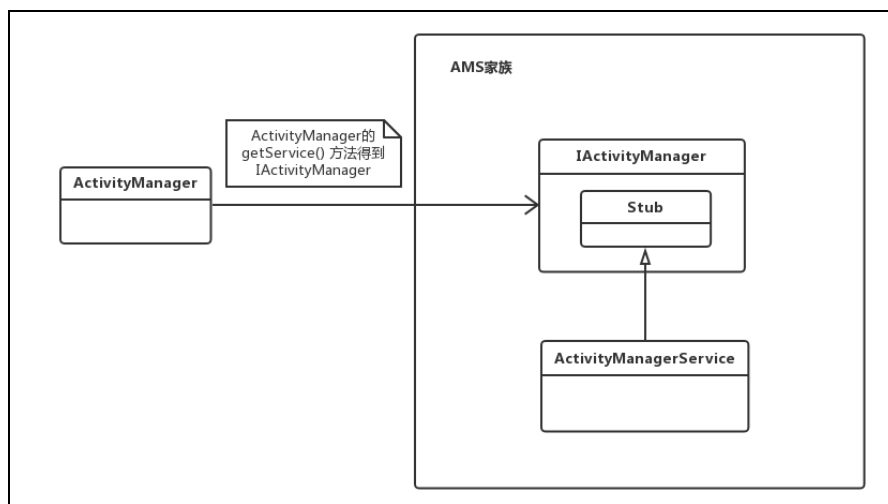


图 6-3 Android 8.0 AMS 家族

对比图 6-3 和图 6-1, 可以发现 Android 8.0 AMS 家族要简单得多, ActivityManager 的 getService 方法会得到 IActivityManager, AMS 只需要继承 IActivityManager.Stub 类, 就可以和 ActivityManager 实现进程间通信了。

6.2 AMS的启动过程

要想更好地理解 AMS, 很有必要了解 AMS 的启动过程, AMS 的启动是在 SystemServer

进程中启动的，关于 SystemServer 进程的启动过程在 2.3 节中已经讲过，这里就从 SystemServer 的 main 方法开始讲起，如下所示：

```
frameworks/base/services/java/com/android/server/SystemServer.java
```

```
public static void main(String[] args) {
    new SystemServer().run();
}
```

main 方法只调用了 SystemServer 的 run 方法，如下所示：

```
frameworks/base/services/java/com/android/server/SystemServer.java
```

```
private void run() {
    try {
        ...
        //创建消息 Looper
        Looper.prepareMainLooper();
        //加载了动态库 libandroid_servers.so
        System.loadLibrary("android_servers");//1
        performPendingShutdown();
        // 创建系统的 Context
        createSystemContext();
        // 创建 SystemServiceManager
        mSystemServiceManager = new SystemServiceManager(mSystemContext);//2
        mSystemServiceManager.setRuntimeRestarted(mRuntimeRestart);
        LocalServices.addService(SystemServiceManager.class, mSystemServiceManager);
        SystemServerInitThreadPool.get();
    } finally {
        traceEnd();
    }
    try {
        traceBeginAndSlog("StartServices");
        //启动引导服务
        startBootstrapServices();//3
        //启动核心服务
        startCoreServices();//4
        //启动其他服务
        startOtherServices();//5
        SystemServerInitThreadPool.shutdown();
    } catch (Throwable ex) {
        Slog.e("System", "*****");
        Slog.e("System", "***** Failure starting system services", ex);
        throw ex;
    } finally {
```

```

        traceEnd();
    }
    ...
}

```

SystemService 的 run 方法已经在 2.3.2 节讲解过，这里再次讲解一遍。在注释 1 处加载了动态库 libandroid_servers.so。接下来在注释 2 处创建 SystemServiceManager，它会对系统的服务进行创建、启动和生命周期管理。在注释 3 处的 startBootstrapServices 方法中用 SystemServiceManager 启动了 ActivityManagerService、PowerManagerService、PackageManagerService 等服务。在注释 4 处的 startCoreServices 方法中则启动了 DropBoxManagerService、BatteryService、UsageStatsService 和 WebViewUpdateService。在注释 5 处的 startOtherServices 方法中启动了 CameraService、AlarmManagerService、VrManagerService 等服务。这些服务的父类均为 SystemService。从注释 3、注释 4、注释 5 处的方法可以看出，官方把系统服务分为了 3 种类型，分别是引导服务、核心服务和其他服务，其中其他服务是一些非紧要和不需要立即启动的服务。我们主要来查看引导服务 AMS 是如何启动的，注释 3 处的 startBootstrapServices 方法如下所示：

```
frameworks/base/services/java/com/android/server/SystemServer.java
```

```

private void startBootstrapServices() {
    ...
    traceBeginAndSlog("StartActivityManager");
    mActivityManagerService = mSystemServiceManager.startService(
        ActivityManagerService.Lifecycle.class).getService();//1
    mActivityManagerService.setSystemServiceManager(mSystemServiceManager);
    mActivityManagerService.setInstaller(installer);
    traceEnd();
    ...
}

```

在注释 1 处调用了 SystemServiceManager 的 startService 方法，该方法的参数是 ActivityManagerService.Lifecycle.class：

```
frameworks/base/services/core/java/com/android/server/SystemServiceManager.java
```

```

public void startService(@NonNull final SystemService service) {
    mServices.add(service);//1
    long time = System.currentTimeMillis();
    try {
        service.onStart();//2
    } catch (RuntimeException ex) {
        throw new RuntimeException("Failed to start service " + service.getClass().
            getName() + ":: onStart threw an exception", ex);
    }
}

```

```

    }
    warnIfTooLong(System.currentTimeMillis() - time, service, "onStart");
}

```

传入的 `SystemService` 类型的 `service` 对象的值为 `ActivityManagerService.Lifecycle.class`。在注释 1 处将 `service` 对象添加到 `ArrayList` 类型的 `mServices` 中来完成注册。在注释 2 处调用 `service` 的 `onStart` 方法来启动 `service` 对象，这个 `service` 对象具体指的是什么呢？我们接着往下看，`Lifecycle` 是 `AMS` 的内部类，代码如下所示：

`frameworks/base/services/core/java/com/android/server/am/ActivityManagerService.java`

```

public static final class Lifecycle extends SystemService {
    private final ActivityManagerService mService;
    public Lifecycle(Context context) {
        super(context);
        mService = new ActivityManagerService(context); //1
    }
    @Override
    public void onStart() {
        mService.start(); //2
    }
    public ActivityManagerService getService() { //3
        return mService;
    }
}

```

上面的代码需要结合 `SystemServiceManager` 的 `startService` 方法来分析。注释 1 处，在 `Lifecycle` 的构造方法中创建了 `AMS` 实例。当调用 `SystemService` 类型的 `service` 的 `onStart` 方法时，实际上是调用了注释 2 处 `AMS` 的 `start` 方法。注释 3 处的 `Lifecycle` 的 `getService` 方法返回 `AMS` 实例，这样我们就知道 `SystemService` 的 `startBootstrapServices` 方法的注释 1 处 `mSystemServiceManager.startService(ActivityManagerService.Lifecycle.class).getService()` 实际得到的就是 `AMS` 实例，`AMS` 的启动过程就讲到这里。

6.3 AMS与应用程序进程

在 2.2.3 节中讲到了 `Zygote` 的 `Java` 框架层中，会创建一个 `Server` 端的 `Socket`，这个 `Socket` 用来等待 `AMS` 请求 `Zygote` 来创建新的应用程序进程。要启动一个应用程序，首先要保证这个应用程序所需要的应用程序进程已经存在。在启动应用程序时 `AMS` 会检查这个应用程序需要的应用程序进程是否存在，不存在就会请求 `Zygote` 进程创建需要的应用程序进程。

这里以 Service 的启动过程为例，来分析 AMS 与应用程序进程的关系。Service 在启动过程中会调用 ActiveServices 的 bringUpServiceLocked 方法，如下所示：

```
frameworks/base/services/core/java/com/android/server/am/ActiveServices.java
```

```
private String bringUpServiceLocked(ServiceRecord r, int intentFlags, boolean
execInFg, boolean whileRestarting, boolean permissionsReviewRequired)
throws TransactionTooLargeException {
    ...
    //获取 Service 想要在哪进程运行
    final String procName = r.processName;//1
    String hostingType = "service";
    ProcessRecord app;
    if (!isolated) {
        app = mAm.getProcessRecordLocked(procName, r.appInfo.uid, false);//2
        if (DEBUG_MU) Slog.v(TAG_MU, "bringUpServiceLocked: appInfo.uid=" + r.
            appInfo.uid + " app=" + app);
        //如果运行 Service 的应用程序进程存在
        if (app != null && app.thread != null) {//3
            try {
                app.addPackage(r.appInfo.packageName, r.appInfo.versionCode, mAm.
                    mProcessStats);
                //启动 Service
                realStartServiceLocked(r, app, execInFg);//4
                return null;
            } catch (TransactionTooLargeException e) {
                throw e;
            } catch (RemoteException e) {
                Slog.w(TAG, "Exception when starting service " + r.shortName, e);
            }
        }
    } else {
        app = r.isolatedProc;
        if (WebViewZygote.isMultiprocessEnabled()
            && r.serviceInfo.packageName.equals(WebViewZygote.getPackageName())) {
            hostingType = "webview_service";
        }
    }
    //如果用来运行 Service 的应用程序进程不存在
    if (app == null && !permissionsReviewRequired) {//5
        //创建应用程序进程
        if ((app=mAm.startProcessLocked(procName, r.appInfo, true, intentFlags,
            hostingType, r.name, false, isolated, false)) == null) {//6
            String msg = "Unable to launch app "
```

```

        + r.appInfo.packageName + "/"
        + r.appInfo.uid + " for service "
        + r.intent.getIntent() + ": process is bad";
    Slog.w(TAG, msg);
    bringDownServiceLocked(r);
    return msg;
}
if (isolated) {
    r.isolatedProc = app;
}
}
...
return null;
}

```

在注释 1 处得到 ServiceRecord 的 processName 的值并赋值给 procName, 其中 processName 用来描述 Service 想要在哪个进程运行, 默认是当前进程, 我们也可以在 AndroidManifest 文件中设置 android:process 属性来新开启一个进程运行 Service。在注释 2 处将 procName 和 Service 的 uid 传入到 AMS 的 getProcessRecordLocked 方法中, 来查询是否存在一个与 Service 对应的 ProcessRecord 类型的对象 app, ProcessRecord 主要用来描述运行的应用程序进程的信息。在注释 5 处判断 Service 对应的 app 为 null 则说明用来运行 Service 的应用程序进程不存在, 则调用注释 6 处的 AMS 的 startProcessLocked 方法来创建对应的应用程序进程, 关于创建应用程序进程请查看第 3 章的内容, 在注释 3 处判断如果用来运行 Service 的应用程序进程存在, 则调用注释 4 处的 realStartServiceLocked 方法来启动 Service, 具体的过程请查看 4.2.2 节。总结一下, AMS 与应用程序进程的关系主要有以下两点:

- 启动应用程序时 AMS 会检查这个应用程序需要的应用程序进程是否存在。
- 如果需要的应用程序进程不存在, AMS 就会请求 Zygote 进程创建需要的应用程序进程。

6.4 AMS重要的数据结构

AMS 涉及了很多数据结构, 这一节我们分析一下 ActivityRecord、TaskRecord 和 ActivityStack, 为什么要学习它们呢? 因为它们和应用开发关联较大, 是 Activity 任务栈模型的基础。

6.4.1 解析ActivityRecord

ActivityRecord 在本书的前几章经常会见到，它内部记录了 Activity 的所有信息，因此它用来描述一个 Activity，它是在启动 Activity 时被创建的，具体是在 ActivityStarter 的 startActivity 方法中被创建的，具体可以查看 4.1.2 节。ActivityRecord 的部分重要成员变量如表 6-1 所示。

表 6-1 ActivityRecord 的部分重要成员变量

名 称	类 型	说 明
service	ActivityManagerService	AMS 的引用
info	ActivityInfo	Activity 中代码和 AndroidManifest 设置的节点信息，比如 launchMode
launchedFromPackage	String	启动 Activity 的包名
taskAffinity	String	Activity 希望归属的栈
task	TaskRecord	ActivityRecord 所在的 TaskRecord
app	ProcessRecord	ActivityRecord 所在的应用程序进程
state	ActivityState	当前 Activity 的状态
icon	int	Activity 的图标资源标识符
theme	int	Activity 的主题资源标识符

从表 6-1 可以看出 ActivityRecord 的作用，其内部存储了 Activity 的所有信息，包括 AMS 的引用、AndroidManifest 节点信息、Activity 状态、Activity 资源信息和 Activity 进程相关信息等，需要注意的是其中含有该 ActivityRecord 所在的 TaskRecord，这就将 ActivityRecord 和 TaskRecord 关联在一起，它们是 Activity 任务栈模型的重要成员，我们接着来查看 TaskRecord。

6.4.2 解析TaskRecord

TaskRecord 用来描述一个 Activity 任务栈，其内部也有很多的成员变量，这里挑出一些重要的成员变量进行介绍，如表 6-2 所示。

表 6-2 TaskRecord 的部分重要成员变量

名 称	类 型	说 明
taskId	int	任务栈的唯一标识符
affinity	String	任务栈的倾向性
intent	Intent	启动这个任务栈的 Intent
mActivities	ArrayList<ActivityRecord>	按照历史顺序排列的 Activity 记录

续表

名 称	类 型	说 明
mStack	ActivityStack	当前归属的 ActivityStack
mService	ActivityManagerService	AMS 的引用

从表 6-2 可以发现 TaskRecord 的作用，其内部存储了任务栈的所有信息，包括任务栈的唯一标识符、任务栈的倾向性、任务栈中的 Activity 记录和 AMS 的引用等，需要注意的是其中含有 ActivityStack，也就是当前 Activity 任务栈所归属的 ActivityStack，我们接着来查看 ActivityStack。

6.4.3 解析ActivityStack

ActivityStack 是一个管理类，用来管理系统所有 Activity，其内部维护了 Activity 的所有状态、特殊状态的 Activity 以及和 Activity 相关的列表等数据。ActivityStack 是由 ActivityStackSupervisor 来进行管理的，而 ActivityStackSupervisor 在 AMS 的构造方法中被创建，如下所示：

```
frameworks/base/services/core/java/com/android/server/am/ActivityManagerService.java
public ActivityManagerService(Context systemContext) {
    ...
    mStackSupervisor = createStackSupervisor();
    ...
}
```

在 createStackSupervisor 方法中创建了 ActivityStackSupervisor：

```
protected ActivityStackSupervisor createStackSupervisor() {
    return new ActivityStackSupervisor(this, mHandler.getLooper());
}
```

1. ActivityStack 的实例类型

在 ActivityStackSupervisor 中有多种 ActivityStack 实例，如下所示：

```
frameworks/base/services/core/java/com/android/server/am/ActivityStackSupervisor.java
public class ActivityStackSupervisor extends ConfigurationContainer implements
DisplayListener {
    ...
    ActivityStack mHomeStack;
    ActivityStack mFocusedStack;
    private ActivityStack mLastFocusedStack;
    ...
}
```


mHomeStack 用来存储 Launcher App 的所有 Activity，mFocusedStack 表示当前正在接收输入或启动下一个 Activity 的所有 Activity。mLastFocusedStack 表示此前接收输入的所有 Activity。通过 ActivityStackSupervisor 提供了获取上述 ActivityStack 的方法，比如要获取 mFocusedStack，只需要调用 ActivityStackSupervisor 的 getFocusedStack 方法就可以了：

```
frameworks/base/services/core/java/com/android/server/am/ActivityStackSupervisor.java
```

```
ActivityStack getFocusedStack() {
    return mFocusedStack;
}
```

2. ActivityState

在 ActivityStack 中通过枚举存储了 Activity 的所有状态，如下所示：

```
frameworks/base/services/core/java/com/android/server/am/ActivityStack.java
```

```
enum ActivityState {
    INITIALIZING,
    RESUMED,
    PAUSING,
    PAUSED,
    STOPPING,
    STOPPED,
    FINISHING,
    DESTROYING,
    DESTROYED
}
```

通过名称我们可以很轻易知道这些状态所代表的意义。应用 ActivityState 的场景会有很多，比如下面的代码：

```
frameworks/base/services/core/java/com/android/server/am/ActivityManagerService.java
```

```
@Override
public void overridePendingTransition(IBinder token, String packageName, int
enterAnim, int exitAnim) {
    synchronized(this) {
        ActivityRecord self = ActivityRecord.isInStackLocked(token);
        if (self == null) {
            return;
        }
        final long origId = Binder.clearCallingIdentity();
        if (self.state == ActivityState.RESUMED
            || self.state == ActivityState.PAUSING) { //1
            mWindowManager.overridePendingAppTransition(packageName,
```

```
        enterAnim, exitAnim, null);
    }
    Binder.restoreCallingIdentity(origId);
}
}
```

overridePendingTransition 方法用于设置 Activity 的切换动画，在注释 1 处可以看到只有 ActivityState 为 RESUMED 状态或者 PAUSING 状态时才会调用 WMS 类型的 mWindowManager 对象的 overridePendingAppTransition 方法来切换动画。

3. 特殊状态的 Activity

在 ActivityStack 中定义了一些特殊状态的 Activity，如下所示：

```
ActivityRecord mPausingActivity = null;//正在暂停的 Activity
ActivityRecord mLastPausedActivity = null;//上一个已经暂停的 Activity
ActivityRecord mLastNoHistoryActivity = null;//最近一次没有历史记录的活动
ActivityRecord mResumedActivity = null;//已经 Resume 的 Activity
ActivityRecord mLastStartedActivity = null;//最近一次启动的 Activity
//传递给 convertToTranslucent 方法的最上层的 Activity
ActivityRecord mTranslucentActivityWaiting = null;
```

这些特殊的状态都是 ActivityRecord 类型的，ActivityRecord 用来记录一个 Activity 的所有信息。

4. 维护的 ArrayList

在 ActivityStack 中维护了很多 ArrayList，这些 ArrayList 中的元素类型主要有 ActivityRecord 和 TaskRecord，如表 6-3 所示。

表 6-3 ArrayList 中的元素类型及其说明

ArrayList	元素类型	说明
mTaskHistory	TaskRecord	所有没有被销毁的 Activity 任务栈
mLRUActivities	ActivityRecord	正在运行的 Activity，列表中的第一个条目是最近最少使用的 Activity
mNoAnimActivities	ActivityRecord	不考虑转换动画的 Activity
mValidateAppTokens	TaskGroup	用于与窗口管理器验证应用令牌

ActivityStack 维护了元素类型为 TaskRecord 的列表，这样 ActivityStack 和 TaskRecord 就有了关联，Activity 任务栈存储在 ActivityStack 中。AMS 重要的数据结构 ActivityRecord、TaskRecord 和 ActivityStack 就讲到这里，要想更多地了解它们请自行阅读源码。

6.5 Activity栈管理

我们平时做应用开发都知道 Activity 是放入在 Activity 任务栈中的，有了任务栈，系统和开发者就能够更好地应用和管理 Activity，来完成各种业务逻辑。这一节我们来学习和 Activity 栈管理相关的知识点，不过在此之前先要了解一下 Activity 任务栈模型。

6.5.1 Activity任务栈模型

Activity 任务栈并不是凭空想象出来的，它是由多种数据结构共同组合而成的，在 6.4 节我们学习了 ActivityRecord、TaskRecord 和 ActivityStack，它们就是 Activity 任务栈模型的重要组成部分，如图 6-4 所示。

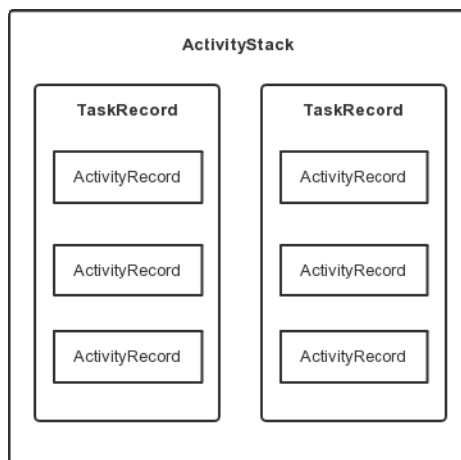


图 6-4 Activity 任务栈模型

ActivityRecord 用来记录一个 Activity 的所有信息，TaskRecord 中包含了一个或多个 ActivityRecord，TaskRecord 用来表示 Activity 的任务栈，用来管理栈中的 ActivityRecord，ActivityStack 又包含了一个或多个 TaskRecord，它是 TaskRecord 的管理者。Activity 栈管理就是建立在 Activity 任务栈模型之上的，有了栈管理，我们可以对应用程序进行操作，应用可以复用自身应用中以及其他应用的 Activity，节省了资源。比如我们使用一款社交应用，这个应用的联系人详情界面提供了联系人的邮箱，当我们点击邮箱时会跳到发送邮件的界面，如图 6-5 所示。

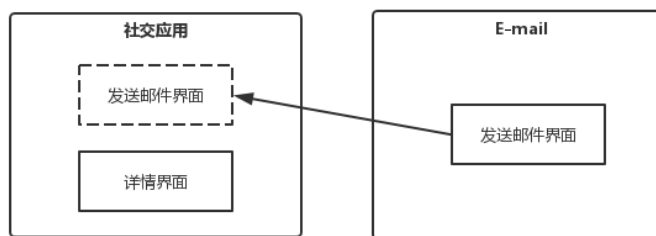


图 6-5 Activity 栈管理

社交应用和系统 E-mail 中的 Activity 是处于不同应用程序进程的，而有了栈管理，就可以把发送邮件界面放到社交应用中详情界面所在栈的栈顶，来做到跨进程操作。为了更灵活地进行栈管理，Android 系统提供了很多配置，包括 Launch Mode、Intent 的 FLAG 和 taskAffinity 等，下面分别对它们进行介绍。

6.5.2 Launch Mode

Launch Mode 大家都不陌生，用于设定 Activity 的启动方式，无论是哪种启动方式，所启动的 Activity 都会位于 Activity 栈的栈顶，主要有以下 4 种 Launch Mode。

- standard：默认模式，每次启动 Activity 都会创建一个新的 Activity 实例。
- singleTop：如果要启动的 Activity 已经在栈顶，则不会重新创建 Activity，同时该 Activity 的 onNewIntent 方法会被调用。如果要启动的 Activity 不在栈顶，则会重新创建该 Activity 的实例。
- singleTask：如果要启动的 Activity 已经存在于它想要归属的栈中，那么不会创建该 Activity 实例，将栈中位于该 Activity 上的所有的 Activity 出栈，同时该 Activity 的 onNewIntent 方法会被调用。如果要启动的 Activity 不存在于它想要归属的栈中，并且该栈存在，则会重新创建该 Activity 的实例。如果要启动的 Activity 想要归属的栈不存在，则首先要创建一个新栈，然后创建该 Activity 实例并压入到新栈中。
- singleInstance：和 singleTask 基本类似，不同的是启动 Activity 时，首先要创建一个新栈，然后创建该 Activity 实例并压入新栈中，新栈中只会存在这一个 Activity 实例。

6.5.3 Intent的FLAG

在 Intent 中定义了很多 FLAG，其中有几个 FLAG 也可以设定 Activity 的启动方式，如果 Launch Mode 和 FLAG 设定的 Activity 的启动方式有冲突，则以 FLAG 设定的为准。

- FLAG_ACTIVITY_SINGLE_TOP: 和 Launch Mode 中的 singleTop 效果是一样的。
- FLAG_ACTIVITY_NEW_TASK: 和 Launch Mode 中的 singleTask 效果是一样的。
- FLAG_ACTIVITY_CLEAR_TOP: 在 Launch Mode 中没有与此对应的模式, 如果要启动的 Activity 已经存在于栈中, 则将所有位于它上面的 Activity 出栈。singleTask 默认具有此标记位的效果。

除了上述这三个 FLAG, 还有一些 FLAG 对我们分析栈管理有些帮助。

- FLAG_ACTIVITY_NO_HISTORY: Activity 一旦退出, 就不会存在于栈中。同样地, 也可以在 AndroidManifest.xml 中设置 android:noHistory。
- FLAG_ACTIVITY_MULTIPLE_TASK: 需要和 FLAG_ACTIVITY_NEW_TASK 一同使用才有效果, 系统会启动一个新的栈来容纳新启动的 Activity。
- FLAG_ACTIVITY_EXCLUDE_FROM_RECENTS: Activity 不会被放入到“最近启动的 Activity”列表中。
- FLAG_ACTIVITY_BROUGHT_TO_FRONT: 这个标志位通常不是由应用程序中的代码设置的, 而是 Launch Mode 为 singleTask 时, 由系统自动加上的。
- FLAG_ACTIVITY_LAUNCHED_FROM_HISTORY: 这个标志位通常不是由应用程序中的代码设置的, 而是从历史记录中启动的 (长按 Home 键调出)。
- FLAG_ACTIVITY_CLEAR_TASK: 需要和 FLAG_ACTIVITY_NEW_TASK 一同使用才有效果, 用于清除与启动的 Activity 相关栈的所有其他 Activity。

接下来通过系统源码来查看 FLAG 的应用, 在 4.1.2 节中讲过根 Activity 启动时会调用 AMS 的 startActivity 方法, 经过层层调用会调用 ActivityStarter 的 startActivityUnchecked 方法, 如图 6-6 所示。

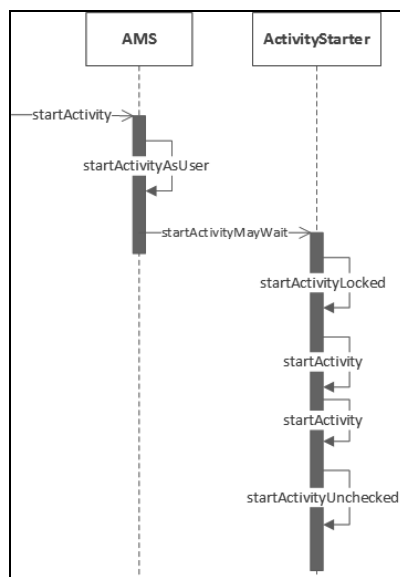


图 6-6 AMS 到 ActivityStarter 的调用过程

```
frameworks/base/services/core/java/com/android/server/am/ActivityStarter.java
```

```
private int startActivityUnchecked(final ActivityRecord r, ActivityRecord
sourceRecord, IVoiceInteractionSession voiceSession, IVoiceInteractor
voiceInteractor, int startFlags, boolean doResume, ActivityOptions options,
```

```

TaskRecord inTask) {
    setInitialState(r, options, inTask, doResume, startFlags, sourceRecord,
        voiceSession, voiceInteractor); //1
    computeLaunchingTaskFlags(); //2
    computeSourceStack();
    mIntent.setFlags(mLaunchFlags); //3
    ...
}

```

在注释 1 处用于初始化启动 Activity 的各种配置，在初始化前会重置各种配置再进行配置，这些配置包括 ActivityRecord、Intent、TaskRecord 和 LaunchFlags（Activity 启动的 FLAG）等。注释 2 处的 computeLaunchingTaskFlags 方法用于计算出 Activity 启动的 FLAG，并将计算的值赋值给 mLaunchFlags。在注释 3 处将 mLaunchFlags 设置给 Intent，达到设定 Activity 的启动方式的目的，接着来查看 computeLaunchingTaskFlags 方法：

```

frameworks/base/services/core/java/com/android/server/am/ActivityStarter.java

private void computeLaunchingTaskFlags() {
    ...
    if (mInTask == null) { //1
        if (mSourceRecord == null) { //2
            if ((mLaunchFlags & FLAG_ACTIVITY_NEW_TASK) == 0 && mInTask == null) { //3
                Slog.w(TAG, "startActivity called from non-Activity context;
                    forcing " + "Intent.FLAG_ACTIVITY_NEW_TASK for: " + mIntent);
                mLaunchFlags |= FLAG_ACTIVITY_NEW_TASK;
            }
        } else if (mSourceRecord.launchMode == LAUNCH_SINGLE_INSTANCE) { //4
            mLaunchFlags |= FLAG_ACTIVITY_NEW_TASK;
        } else if (mLaunchSingleInstance || mLaunchSingleTask) { //5
            mLaunchFlags |= FLAG_ACTIVITY_NEW_TASK;
        }
    }
}

```

计算启动的 FLAG 的逻辑比较复杂，这里只截取了一小部分，当注释 1 处的 TaskRecord 类型的 mInTask 为 null 时，说明 Activity 要加入的栈不存在，因此，这一小段代码主要解决的问题就是 Activity 要加入的栈不存在时如何计算出启动的 FLAG。在注释 2 处，ActivityRecord 类型的 mSourceRecord 用于描述“初始 Activity”，什么是“初始 Activity”呢？比如 ActivityA 启动了 ActivityB，ActivityA 就是初始 Activity。同时满足注释 2 和注释 3 的条件则需要创建一个新栈。在注释 4 处，如果“初始 Activity”所在的栈只允许有一个 Activity 实例，则也需要创建一个新栈。在注释 5 处，如果 Launch Mode 设置了 singleTask 或 singleInstance，则也要创建一个新栈。

6.5.4 taskAffinity

我们可以在 AndroidManifest.xml 中设置 android:taskAffinity，用来指定 Activity 希望归属的栈，在默认情况下，同一个应用程序的所有的 Activity 都有着相同的 taskAffinity。taskAffinity 在下面两种情况时会产生效果。

(1) taskAffinity 与 FLAG_ACTIVITY_NEW_TASK 或者 singleTask 配合。如果新启动 Activity 的 taskAffinity 和栈的 taskAffinity 相同则加入到该栈中；如果不同，就会创建新栈。

(2) taskAffinity 与 allowTaskReparenting 配合。如果 allowTaskReparenting 为 true，说明 Activity 具有转移的能力。拿此前的邮件为例（图 6-5），当社交应用启动了发送邮件的 Activity，此时发送邮件的 Activity 是和社交应用处于同一个栈中的，并且这个栈位于前台。如果发送邮件的 Activity 的 allowTaskReparenting 设置为 true，此后 E-mail 应用所在的栈位于前台时，发送邮件的 Activity 就会由社交应用的栈中转移到与它更亲近的邮件应用（taskAffinity 相同）所在的栈中，如图 6-7 所示。

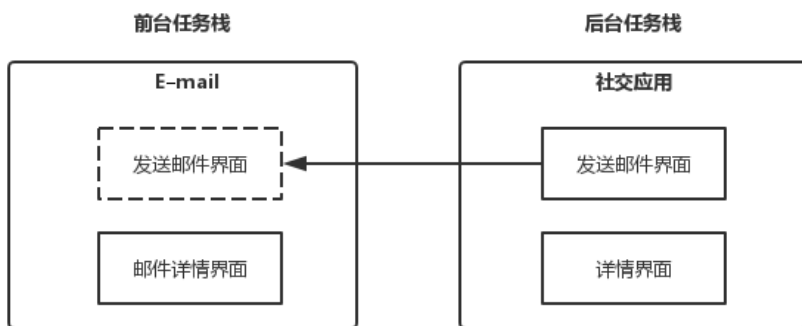


图 6-7 任务栈转移

接着通过系统源码来查看 taskAffinity 的应用。ActivityStackSupervisor 的 findTaskLocked 方法用于找到 Activity 最匹配的栈，最终会调用 ActivityStack 的 findTaskLocked 方法：

```
frameworks/base/services/core/java/com/android/server/am/ActivityStack.java
```

```
void findTaskLocked(ActivityRecord target, FindTaskResult result) {
    ...
    for (int taskNdx = mTaskHistory.size() - 1; taskNdx >= 0; --taskNdx) { //1
        final TaskRecord task = mTaskHistory.get(taskNdx); //2
        ...
        else if (!isDocument && !taskIsDocument
            && result.r == null && task.canMatchRootAffinity()) {
            if (task.rootAffinity.equals(target.taskAffinity)) { //3
```

```

        if (DEBUG_TASKS) Slog.d(TAG_TASKS, "Found matching affinity candidate!");
        result.r = r;
        result.matchedByRootAffinity = true;
    }
    } else if (DEBUG_TASKS) Slog.d(TAG_TASKS, "Not a match: " + task);
}
}

```

这个方法的逻辑比较复杂，这里截取了和 `taskAffinity` 相关的部分。在注释 1 处遍历 `mTaskHistory` 列表，列表的元素为 `TaskRecord`，它用于存储没有被销毁的栈。在注释 2 处得到某一个栈的信息。在注释 3 处将栈的 `rootAffinity`（初始的 `taskAffinity`）和目标 `Activity` 的 `taskAffinity` 做对比，如果相同，则将 `FindTaskResult` 的 `matchedByRootAffinity` 属性设置为 `true`，说明找到了匹配的栈。

6.6 本章小结

本章介绍了 AMS 的家族、AMS 的启动、AMS 重要的数据结构和 Activity 栈管理等知识点，关于 AMS 有很多内容可以学习，本章也只是介绍了和应用开发有所关联的部分，如果想要更多地了解 AMS 的原理则需要大家自行去阅读源码并做总结。另外本章和前面 5 章的内容有所关联，如果前面 5 章你理解并掌握了，那么读完这一章你可能会有更多的发现。

第 7 章

理解 WindowManager

WindowManagerService (WMS) 和 AMS 一样，都是 Android 应用开发需要掌握的知识点。WMS 也很复杂并且功能繁多，需要花很多篇幅来进行讲解。如果直接讲解 WMS 会很难理解，为了更好地理解 WMS，需要先了解 WindowManager 的相关知识，这是因为从应用开发角度来看，WindowManager 是与 WMS 关联最紧密的类。这一章我们会介绍 WindowManager 体系、Window 的属性和 Window 的操作。

7.1 Window、WindowManager和WMS

Window 我们应该很熟悉，它是一个抽象类，具体的实现类为 PhoneWindow，它对 View 进行管理。WindowManager 是一个接口类，继承自接口 ViewManager，从名称就知道它是用来管理 Window 的，它的实现类为 WindowManagerImpl。如果我们想要对 Window (View) 进行添加、更新和删除操作就可以使用 WindowManager，WindowManager 会将具体的工作交由 WMS 来处理，WindowManager 和 WMS 通过 Binder 来进行跨进程通信，WMS 作为系统服务有很多 API 是不会暴露给 WindowManager 的，这一点与 ActivityManager 和 AMS 的关系有些类似。

关于 WMS 的功能，会在第 8 章进行介绍，这里我们只需要知道它的主要功能是管理 Window 就可以了。Window、WindowManager 和 WMS 的关系可以简略地用图 7-1 来表示。

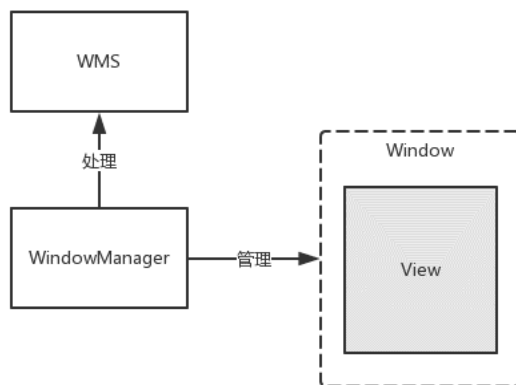


图 7-1 Window、WindowManager 和 WMS 的关系

Window 包含了 View 并对 View 进行管理，Window 用虚线来表示是因为 Window 是一个抽象概念，用来描述一个窗口，并不是真实存在的，Window 的实体其实也是 View。WindowManager 用来管理 Window，而 WindowManager 所提供的功能最终会由 WMS 进行处理。

7.2 WindowManager的关联类

接下来从源码角度分析 WindowManager 的关联类，从中我们可以更好地理解 Window 和 WindowManager 的关系。

WindowManager 是一个接口类，继承自接口 ViewManager，ViewManager 中定义了 3 个方法，分别用来添加、更新和删除 View，如下所示：

frameworks/base/core/java/android/view/ViewManager.java

```

public interface ViewManager
{
    public void addView(View view, ViewGroup.LayoutParams params);
    public void updateViewLayout(View view, ViewGroup.LayoutParams params);
    public void removeView(View view);
}

```

WindowManager 也继承了这些方法，而这些方法传入的参数都是 View 类型，说明 Window 是以 View 的形式存在的。WindowManager 在继承 ViewManager 的同时，又加入很多功能，包括 Window 的类型和层级相关的常量、内部类以及一些方法，其中有两个方法

是根据 Window 的特性加入的，如下所示：

```
public Display getDefaultDisplay();
public void removeViewImmediate(View view);
```

getDefaultDisplay 方法能够得知这个 WindowManager 实例将 Window 添加到哪个屏幕上了，换句话说，就是得到 WindowManager 所管理的屏幕 (Display)。removeViewImmediate 方法则规定在这个方法返回前要立即执行 View.onDetachedFromWindow()，来完成传入的 View 相关的销毁工作。

Window 是一个抽象类，它的具体实现类为 PhoneWindow，PhoneWindow 是何时创建的呢？在 Activity 启动过程中会调用 ActivityThread 的 performLaunchActivity 方法，performLaunchActivity 方法中又会调用 Activity 的 attach 方法，PhoneWindow 就是在 Activity 的 attach 方法中创建的，如下所示：

frameworks/base/core/java/android/app/Activity.java

```
final void attach(Context context, ActivityThread aThread,
    Instrumentation instr, IBinder token, int ident,
    Application application, Intent intent, ActivityInfo info,
    CharSequence title, Activity parent, String id,
    NonConfigurationInstances lastNonConfigurationInstances,
    Configuration config, String referrer, IVoiceInteractor voiceInteractor,
    Window window, ActivityConfigCallback activityConfigCallback) {
    attachBaseContext(context);
    mFragments.attachHost(null /*parent*/);
    mWindow = new PhoneWindow(this, window, activityConfigCallback); //1
    ...
    /**
    *2
    */
    mWindow.setWindowManager((WindowManager)context.getSystemService
        (Context.WINDOW_SERVICE), mToken, mComponent.flattenToString(),
        (info.flags & ActivityInfo.FLAG_HARDWARE_ACCELERATED) != 0);
    if (mParent != null) {
        mWindow.setContainer(mParent.getWindow());
    }
    mWindowManager = mWindow.getWindowManager();
    mCurrentConfig = config;
    mWindow.setColorMode(info.colorMode);
}
```

在注释 1 处创建了 PhoneWindow，在注释 2 处调用了 PhoneWindow 的 setWindowManager

方法，这个方法在 PhoneWindow 的父类 Window 中实现。

frameworks/base/core/java/android/view/Window.java

```
public void setWindowManager(WindowManager wm, IBinder appToken, String appName,
    boolean hardwareAccelerated) {
    mAppToken = appToken;
    mAppName = appName;
    mHardwareAccelerated = hardwareAccelerated
        || SystemProperties.getBoolean(PROPERTY_HARDWARE_UI, false);
    if (wm == null) {
        wm = (WindowManager)mContext.getSystemService(Context.WINDOW_SERVICE); //1
    }
    mWindowManager = ((WindowManagerImpl)wm).createLocalWindowManager(this); //2
}
```

如果传入的 WindowManager 为 null，就会在注释 1 处调用 Context 的 getSystemService 方法，并传入服务的名称 Context.WINDOW_SERVICE (值为 window)，具体在 ContextImpl 中实现，如下所示：

frameworks/base/core/java/android/app/ContextImpl.java

```
@Override
public Object getSystemService(String name) {
    return SystemServiceRegistry.getSystemService(this, name);
}
```

在 getSystemService 方法中会调用 SystemServiceRegistry 的 getSystemServiceName 方法：

frameworks/base/core/java/android/app/SystemServiceRegistry.java

```
public static String getSystemServiceName(Class<?> serviceClass) {
    return SYSTEM_SERVICE_NAMES.get(serviceClass);
}
```

SYSTEM_SERVICE_NAMES 是一个 HashMap 类型的数据，它用来存储服务的名称，那么传入的 Context.WINDOW_SERVICE 到底对应着什么？我们接着往下看：

frameworks/base/core/java/android/app/SystemServiceRegistry.java

```
final class SystemServiceRegistry {
    ...
    private SystemServiceRegistry() { }
    static {
        ...
        registerService(Context.WINDOW_SERVICE, WindowManager.class,
            new CachedServiceFetcher<WindowManager>() {
```

```

@Override
public WindowManager createService(ContextImpl ctx) {
    return new WindowManagerImpl(ctx); //1
}});
...
}
}

```

在 `SystemServiceRegistry` 的静态代码块中会调用多个 `registerService` 方法, 这里只列举了和本节有关的一个。 `registerService` 方法内部会将传入的服务的名称存入到 `SYSTEM_SERVICE_NAMES` 中。从注释 1 处可以看出, 传入的 `Context.WINDOW_SERVICE` 对应的就是 `WindowManagerImpl` 实例, 因此得出结论, `Context` 的 `getSystemService` 方法得到的是 `WindowManagerImpl` 实例。我们再回到 `Window` 的 `setWindowManager` 方法, 在注释 1 处得到 `WindowManagerImpl` 实例后转为 `WindowManager` 类型, 在注释 2 处调用了 `WindowManagerImpl` 的 `createLocalWindowManager` 方法:

```
frameworks/base/core/java/android/view/WindowManagerImpl.java
```

```

public WindowManagerImpl createLocalWindowManager(Window parentWindow) {
    return new WindowManagerImpl(mContext, parentWindow);
}

```

`createLocalWindowManager` 方法同样也是创建 `WindowManagerImpl`, 不同的是这次创建 `WindowManagerImpl` 时将创建它的 `Window` 作为参数传了进来, 这样 `WindowManagerImpl` 就持有了 `Window` 的引用, 可以对 `Window` 进行操作, 比如在 `Window` 中添加 `View`, 会调用 `WindowManagerImpl` 的 `addView` 方法, 如下所示:

```
frameworks/base/core/java/android/view/WindowManagerImpl.java
```

```

@Override
public void addView(@NonNull View view, @NonNull ViewGroup.LayoutParams params)
{
    applyDefaultToken(params);
    mGlobal.addView(view, params, mContext.getDisplay(), mParentWindow); //1
}

```

在注释 1 处调用了 `WindowManagerGlobal` 的 `addView` 方法, 其中最后一个参数 `mParentWindow` 就是上面提到的 `Window`, 可以看出 `WindowManagerImpl` 虽然是 `WindowManager` 的实现类, 但是没有实现什么功能, 而是将功能实现委托给了 `WindowManagerGlobal`, 这里用到的是桥接模式。关于 `WindowManagerGlobal` 的 `addView` 方法会在 7.4 节中进行介绍。我们来查看在 `WindowManagerImpl` 中是如何定义 `WindowManagerGlobal` 的, 如下所示:

frameworks/base/core/java/android/view/WindowManagerImpl.java

```
public final class WindowManagerImpl implements WindowManager {
    private final WindowManagerGlobal mGlobal = WindowManagerGlobal.getInstance();//1
    private final Context mContext;
    private final Window mParentWindow;//2
    ...
    private WindowManagerImpl(Context context, Window parentWindow) {
        mContext = context;
        mParentWindow = parentWindow;//3
    }
    ...
}
```

在注释 1 处可以看出 WindowManagerGlobal 是一个单例，说明在一个进程中只有一个 WindowManagerGlobal 实例。注释 2 处的代码结合注释 3 处的代码说明这个 WindowManagerImpl 实例会作为哪个 Window 的子 Window，这也就说明在一个进程中 WindowManagerImpl 可能会有多个实例。

通过如上的源码分析，WindowManager 的关联类如图 7-2 所示。

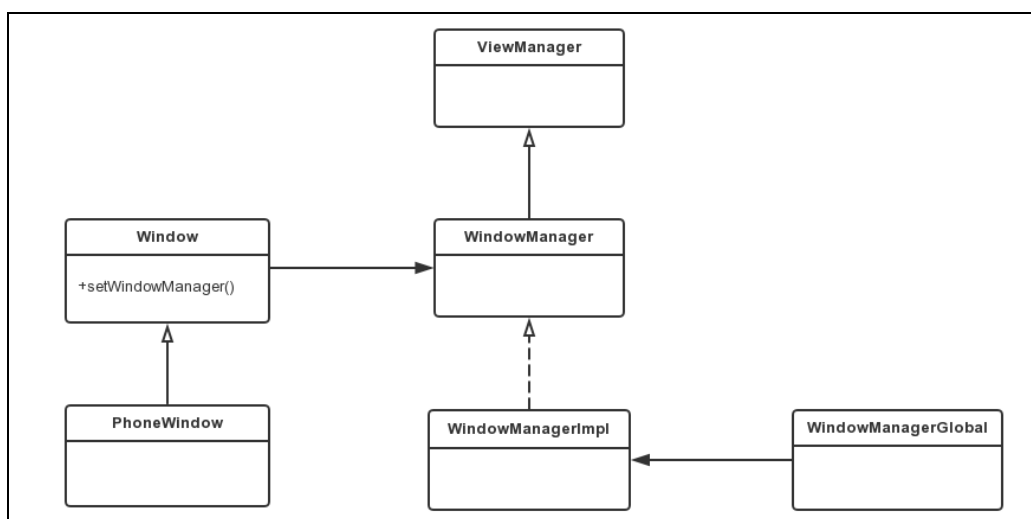


图 7-2 WindowManager 的关联类

从图 7-2 可以看出，PhoneWindow 继承自 Window，Window 通过 setWindowManager 方法与 WindowManager 发生关联。WindowManager 继承自接口 ViewManager，WindowManagerImpl 是 WindowManager 接口的实现类，但是具体的功能都会委托给 WindowManagerGlobal 来实现。

7.3 Window的属性

在 7.2 节中我们讲过了 Window、WindowManager 和 WMS 之间的关系，WMS 是 Window 的最终管理者，Window 好比是员工，WMS 是老板，为了方便老板管理员工则需要定义一些“协议”，这些“协议”就是 Window 的属性，它们被定义在 WindowManager 的内部类 LayoutParams 中，了解 Window 的属性能够更好地理解 WMS 的内部原理。Window 的属性有很多种，与应用开发最密切的有 3 种，它们分别是 Type（Window 的类型）、Flag（Window 的标志）和 SoftInputMode（软键盘相关模式），下面分别介绍这 3 种 Window 的属性。

7.3.1 Window的类型和显示次序

Window 的类型有很多种，比如应用程序窗口、系统错误窗口、输入法窗口、PopupWindow、Toast、Dialog 等。总的来说 Window 分为三大类型，分别是 Application Window（应用程序窗口）、Sub Window（子窗口）、System Window（系统窗口），每个大类型中又包含了很多种类型，它们都定义在 WindowManager 的静态内部类 LayoutParams 中，接下来分别对这三大类型进行讲解。

1. 应用程序窗口

Activity 就是一个典型的应用程序窗口，应用程序窗口包含的类型如下所示：

```
frameworks/base/core/java/android/view/WindowManager.java
```

```
public static final int FIRST_APPLICATION_WINDOW = 1;//1
//窗口的基础值，其他的窗口值要大于这个值
public static final int TYPE_BASE_APPLICATION = 1;
public static final int TYPE_APPLICATION = 2;//普通的应用程序窗口类型
//应用程序启动窗口类型，用于系统在应用程序窗口启动前显示的窗口
public static final int TYPE_APPLICATION_STARTING = 3;
public static final int TYPE_DRAWN_APPLICATION = 4;
public static final int LAST_APPLICATION_WINDOW = 99;//2
```

应用程序窗口共包含了以上几种 Type 值，其中注释 1 处的 Type 表示应用程序窗口类型初始值，注释 2 处的 Type 表示应用程序窗口类型结束值，也就是说应用程序窗口的 Type 值范围为 1~99，这个数值的大小涉及窗口的层级，后面会讲到。

2. 子窗口

子窗口，顾名思义，它不能独立存在，需要附着在其他窗口才可以，PopupWindow 就属于子窗口。子窗口的类型定义如下所示：

```
public static final int FIRST_SUB_WINDOW = 1000; //子窗口类型初始值
public static final int TYPE_APPLICATION_PANEL = FIRST_SUB_WINDOW;
public static final int TYPE_APPLICATION_MEDIA = FIRST_SUB_WINDOW + 1;
public static final int TYPE_APPLICATION_SUB_PANEL = FIRST_SUB_WINDOW + 2;
public static final int TYPE_APPLICATION_ATTACHED_DIALOG = FIRST_SUB_WINDOW + 3;
public static final int TYPE_APPLICATION_MEDIA_OVERLAY = FIRST_SUB_WINDOW + 4;
public static final int TYPE_APPLICATION_ABOVE_SUB_PANEL = FIRST_SUB_WINDOW + 5;
public static final int LAST_SUB_WINDOW = 1999; //子窗口类型结束值
```

可以看出子窗口的 Type 值范围为 1000~1999。

3. 系统窗口

Toast、输入法窗口、系统音量条窗口、系统错误窗口都属于系统窗口。系统窗口的类型定义如下所示：

```
public static final int FIRST_SYSTEM_WINDOW = 2000; //系统窗口类型初始值
public static final int TYPE_STATUS_BAR = FIRST_SYSTEM_WINDOW; //系统状态栏窗口
public static final int TYPE_SEARCH_BAR = FIRST_SYSTEM_WINDOW+1; //搜索条窗口
public static final int TYPE_PHONE = FIRST_SYSTEM_WINDOW+2; //通话窗口
public static final int TYPE_SYSTEM_ALERT = FIRST_SYSTEM_WINDOW+3; //系统ALERT窗口
public static final int TYPE_KEYGUARD = FIRST_SYSTEM_WINDOW+4; //锁屏窗口
public static final int TYPE_TOAST = FIRST_SYSTEM_WINDOW+5; //TOAST窗口
...
public static final int LAST_SYSTEM_WINDOW = 2999; //系统窗口类型结束值
```

系统窗口的类型值有接近 40 个，这里只列出了一小部分，系统窗口的 Type 值范围为 2000~2999。

4. 窗口显示次序

当一个进程向 WMS 申请一个窗口时，WMS 会为窗口确定显示次序。为了方便窗口显示次序的管理，手机屏幕可以虚拟地用 X、Y、Z 轴来表示，其中 Z 轴垂直于屏幕，从屏幕内指向屏幕外，这样确定窗口显示次序也就是确定窗口在 Z 轴上的次序，这个次序称为 Z-Oder。Type 值是 Z-Oder 排序的依据，我们知道应用程序窗口的 Type 值范围为 1~99，子窗口 1000~1999，系统窗口 2000~2999，在一般情况下，Type 值越大则 Z-Oder 排序越靠前，就越靠近用户。当然窗口显示次序的逻辑不会这么简单，情况会比较多，举个常见的情况：当多个窗口的 Type 值都是 TYPE_APPLICATION，这时 WMS 会结合各种情况给出最终的 Z-Oder，这个逻辑不在本书的讨论范围，这里我们只需要知道窗口显示次序的基本规则就可以了。

7.3.2 Window的标志

Window 的标志也就是 Flag, 用于控制 Window 的显示, 同样被定义在 WindowManager 的内部类 LayoutParams 中, 一共有 20 多个, 这里给出几个比较常用的, 如表 7-1 所示。

表 7-1 常用的 Window 的标志

Flag	描 述
FLAG_ALLOW_LOCK_WHILE_SCREEN_ON	只要窗口可见, 就允许在开启状态的屏幕上锁屏
FLAG_NOT_FOCUSABLE	窗口不能获得输入焦点, 设置该标志的同时, FLAG_NOT_TOUCH_MODAL 也会被设置
FLAG_NOT_TOUCHABLE	窗口不接收任何触摸事件
FLAG_NOT_TOUCH_MODAL	将该窗口区域外的触摸事件传递给其他的 Window, 而自己只会处理窗口区域内的触摸事件
FLAG_KEEP_SCREEN_ON	只要窗口可见, 屏幕就会一直亮着
FLAG_LAYOUT_NO_LIMITS	允许窗口超过屏幕之外
FLAG_FULLSCREEN	隐藏所有的屏幕装饰窗口, 比如在游戏、播放器中的全屏显示
FLAG_SHOW_WHEN_LOCKED	窗口可以在锁屏的窗口之上显示
FLAG_IGNORE_CHEEK_PRESSES	当用户的脸贴近屏幕时 (比如打电话), 不会去响应此事件
FLAG_TURN_SCREEN_ON	窗口显示时将屏幕点亮

设置 Window 的 Flag 有 3 种方法, 第一种是通过 Window 的 addFlags 方法:

```
Window mWindow =getWindow();
mWindow.addFlags(WindowManager.LayoutParams.FLAG_FULLSCREEN);
```

第二种通过 Window 的 setFlags 方法:

```
Window mWindow =getWindow();
mWindow.setFlags(WindowManager.LayoutParams.FLAG_FULLSCREEN,
WindowManager.LayoutParams.FLAG_FULLSCREEN);
```

其实 Window 的 addFlags 方法内部会调用 setFlags 方法, 因此这两种方法区别不大。第三种则是给 LayoutParams 设置 Flag, 并通过 WindowManager 的 addView 方法进行添加, 如下所示:

```
WindowManager.LayoutParams mWindowLayoutParams =
    new WindowManager.LayoutParams();
mWindowLayoutParams.flags=WindowManager.LayoutParams.FLAG_FULLSCREEN;
WindowManager mWindowManager =(WindowManager) getSystemService(Context.
WINDOW_SERVICE);
TextView mTextView=new TextView(this);
mWindowManager.addView(mTextView,mWindowLayoutParams);
```

7.3.3 软键盘相关模式

窗口和窗口的叠加是十分常见的场景，但如果其中的窗口是软键盘窗口，可能就会出现一些问题，比如典型的用户登录界面，默认的情况弹出的软键盘窗口可能会盖住输入框下方的按钮，这样用户体验会非常糟糕。为了使得软键盘窗口能够按照期望来显示，WindowManager 的静态内部类 LayoutParams 中定义了软键盘相关模式，这里给出常用的几个，如表 7-2 所示。

表 7-2 软键盘相关模式

SoftInputMode	描 述
SOFT_INPUT_STATE_UNSPECIFIED	没有指定状态，系统会选择一个合适的状态或依赖于主题的设置
SOFT_INPUT_STATE_UNCHANGED	不会改变软键盘状态
SOFT_INPUT_STATE_HIDDEN	当用户进入该窗口时，软键盘默认隐藏
SOFT_INPUT_STATE_ALWAYS_HIDDEN	当窗口获取焦点时，软键盘总是被隐藏
SOFT_INPUT_ADJUST_RESIZE	当软键盘弹出时，窗口会调整大小
SOFT_INPUT_ADJUST_PAN	当软键盘弹出时，窗口不需要调整大小，要确保输入焦点是可见的

从上面给出的 SoftInputMode，可以发现，它们与 AndroidManifest 中 Activity 的属性 android:windowSoftInputMode 是对应的。因此，除了在 AndroidManifest 中为 Activity 设置 android:windowSoftInputMode 以外还可以在 Java 代码中为 Window 设置 SoftInputMode，如下所示：

```
getWindow().setSoftInputMode(WindowManager.LayoutParams.SOFT_INPUT_ADJUST_RESIZE);
```

7.4 Window 的操作

WindowManager 对 Window 进行管理，说到管理那就离不开对 Window 的添加、更新和删除的操作，在这里我们把它们统称为 Window 的操作。对于 Window 的操作，最终都是交由 WMS 来进行处理的。窗口的操作分为两大部分，一部分是 WindowManager 处理部分，另一部分是 WMS 处理部分。我们知道 Window 分为三大类，分别是 Application Window（应用程序窗口）、Sub Window（子窗口）和 System Window（系统窗口），对于不同类型的窗口添加过程会有所不同，但是对于 WMS 处理部分，添加的过程基本上是一样的，WMS 对于这三大类的窗口基本是“一视同仁”的，如图 7-3 所示。

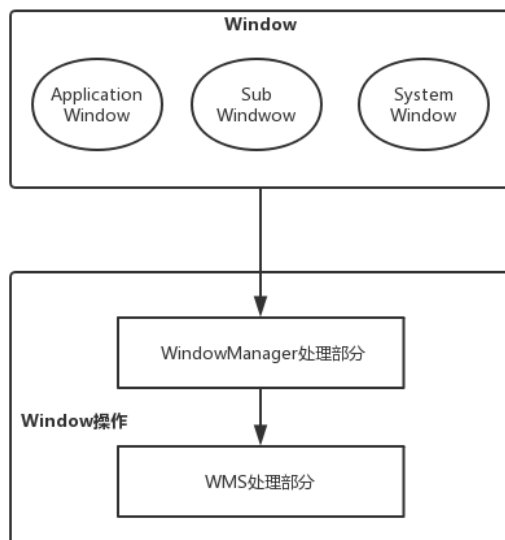


图 7-3 Window 的操作

本节主要讲解 Window 操作的 WindowManager 处理部分，至于 WMS 处理部分会在第 8 章进行讲解。

7.4.1 系统窗口的添加过程

我们知道 Window 分为三大类型，不同类型的 Window 的添加过程也不尽相同，本节主要讲解系统窗口的添加过程。系统窗口的添加过程也会根据不同的系统窗口有所区别，这里以系统窗口 StatusBar 为例，StatusBar 是 SystemUI 的重要组成部分，具体就是指系统状态栏，用于显示时间、电量和信号等信息。我们来查看 StatusBar 的 addStatusBarWindow 方法，这个方法负责为 StatusBar 添加 Window，如下所示：

```
frameworks/base/packages/SystemUI/src/com/android/systemui/statusbar/phone/StatusBar.java
```

```
private void addStatusBarWindow() {
    makeStatusBarView();//1
    mStatusBarWindowManager = Dependency.get(StatusBarWindowManager.class);
    mRemoteInputController = new RemoteInputController(mHeadsUpManager);
    mStatusBarWindowManager.add(mStatusBarWindow, getStatusBarHeight());//2
}
```

在注释 1 处用于构建 StatusBar 的视图。在注释 2 处调用了 StatusBarWindowManager 的 add 方法，并将 StatusBar 的视图 (StatusBarWindowView) 和 StatusBar 的高度传进去，StatusBarWindowManager 的 add 方法如下所示：

```
frameworks/base/packages/SystemUI/src/com/android/systemui/statusbar/phone/StatusBarWindowMa
nager.java
```

```
public void add(View statusBarView, int barHeight) {
    mLP = new WindowManager.LayoutParams(
        ViewGroup.LayoutParams.MATCH_PARENT,
        barHeight,
        WindowManager.LayoutParams.TYPE_STATUS_BAR, //1
        WindowManager.LayoutParams.FLAG_NOT_FOCUSABLE
            | WindowManager.LayoutParams.FLAG_TOUCHABLE_WHEN_WAKING
            | WindowManager.LayoutParams.FLAG_SPLIT_TOUCH
            | WindowManager.LayoutParams.FLAG_WATCH_OUTSIDE_TOUCH
            | WindowManager.LayoutParams.FLAG_DRAWS_SYSTEM_BAR_BACKGROUNDS,
        PixelFormat.TRANSLUCENT);
    mLP.token = new Binder();
    mLP.flags |= WindowManager.LayoutParams.FLAG_HARDWARE_ACCELERATED;
    mLP.gravity = Gravity.TOP;
    mLP.softInputMode = WindowManager.LayoutParams.SOFT_INPUT_ADJUST_RESIZE;
    mLP.setTitle("StatusBar");
    mLP.packageName = mContext.getPackageName();
    mStatusBarView = statusBarView;
    mBarHeight = barHeight;
    mWindowManager.addView(mStatusBarView, mLP); //2
    mLPChanged = new WindowManager.LayoutParams();
    mLPChanged.copyFrom(mLP);
}
```

首先通过创建 LayoutParams 来配置 StatusBar 视图的属性，包括 Width、Height、Type、Flag、Gravity、SoftInputMode 等。关键在注释 1 处，设置了 TYPE_STATUS_BAR，表示 StatusBar 视图的窗口类型是状态栏。在注释 2 处调用了 WindowManager 的 addView 方法，addView 方法定义在 WindowManager 的父类接口 ViewManager 中，而 addView 方法则是在 WindowManagerImpl 中实现的，如下所示：

```
frameworks/base/core/java/android/WindowManagerImpl.java
```

```
@Override
public void addView(@NonNull View view, @NonNull ViewGroup.LayoutParams params) {
    applyDefaultToken(params);
    mGlobal.addView(view, params, mContext.getDisplay(), mParentWindow);
}
```

addView 方法的第一个参数的类型为 View，说明窗口也是以 View 的形式存在的。addView 方法中会调用 WindowManagerGlobal 的 addView 方法，如下所示：

```
frameworks/base/core/java/android/view/WindowManagerGlobal.java
```

```
public void addView(View view, ViewGroup.LayoutParams params,
    Display display, Window parentWindow) {

    ...省略参数检查
    final WindowManager.LayoutParams wparams = (WindowManager.LayoutParams)
        params;
    if (parentWindow != null) {
        parentWindow.adjustLayoutParamsForSubWindow(wparams); //1
    } else {
        final Context context = view.getContext();
        if (context != null
            && (context.getApplicationInfo().flags
                & ApplicationInfo.FLAG_HARDWARE_ACCELERATED) != 0) {
            wparams.flags |= WindowManager.LayoutParams.FLAG_HARDWARE_ACCELERATED;
        }
    }
    ViewRootImpl root;
    View panelParentView = null;
    synchronized (mLock) {
        ...
        root = new ViewRootImpl(view.getContext(), display); //2
        view.setLayoutParams(wparams);
        mViews.add(view); //3
        mRoots.add(root); //4
        mParams.add(wparams); //5
        try {
            root.setView(view, wparams, panelParentView); //6
        } catch (RuntimeException e) {
            if (index >= 0) {
                removeViewLocked(index, true);
            }
            throw e;
        }
    }
}
```

在介绍 `addView` 方法前我们首先要了解 `WindowManagerGlobal` 中维护的和 `Window` 操作相关的 3 个列表，在窗口的添加、更新和删除过程中都会涉及这 3 个列表，它们分别是 `View` 列表（`ArrayList<View> mViews`）、布局参数列表（`ArrayList<WindowManager.LayoutParams> mParams`）和 `ViewRootImpl` 列表（`ArrayList<ViewRootImpl> mRoots`）。了解了这 3 个列表后，我们接着分析 `addView` 方法，首先会对参数 `view`、`params` 和 `display` 进

行检查。在注释 1 处，如果当前窗口要作为子窗口，就会根据父窗口对子窗口的 `WindowManager.LayoutParams` 类型的 `wparams` 对象进行相应调整。在注释 3 处将添加的 `View` 保存到 `View` 列表中。在注释 5 处将窗口的参数保存到布局参数列表中。在注释 2 处创建了 `ViewRootImp` 并赋值给 `root`，紧接着在注释 4 处将 `root` 存入到 `ViewRootImpl` 列表中。在注释 6 处将窗口和窗口的参数通过 `setView` 方法设置到 `ViewRootImpl` 中，可见我们添加窗口这一操作是通过 `ViewRootImpl` 来进行的。`ViewRootImpl` 身负了很多职责，主要有以下几点：

- `View` 树的根并管理 `View` 树。
- 触发 `View` 的测量、布局和绘制。
- 输入事件的中转站。
- 管理 `Surface`。
- 负责与 `WMS` 进行进程间通信。

了解了 `ViewRootImpl` 的职责后，我们接着来查看 `ViewRootImpl` 的 `setView` 方法：

```
frameworks/base/core/java/android/view/ViewRootImpl.java
```

```
public void setView(View view, WindowManager.LayoutParams attrs, View panelParentView) {
    synchronized (this) {
        ...
        try {
            mOrigWindowType = mWindowAttributes.type;
            mAttachInfo.mRecomputeGlobalAttributes = true;
            collectViewAttributes();
            res = mWindowSession.addToDisplay(mWindow, mSeq, mWindowAttributes,
                getHostVisibility(), mDisplay.getDisplayId(),
                mAttachInfo.mContentInsets, mAttachInfo.mStableInsets,
                mAttachInfo.mOutsets, mInputChannel);
        }
        ...
    }
}
```

在 `setView` 方法中有很多逻辑，这里只截取了一小部分，主要就是调用了 `mWindowSession` 的 `addToDisplay` 方法，`mWindowSession` 是 `IWindowSession` 类型的，它是一个 `Binder` 对象，用于进行进程间通信，`IWindowSession` 是 `Client` 端的代理，它的 `Server` 端的实现为 `Session`，此前的代码逻辑都是运行在本地进程的，而 `Session` 的 `addToDisplay` 方法则运行在 `WMS` 所在的进程（`SystemServer` 进程）中，如图 7-4 所示。

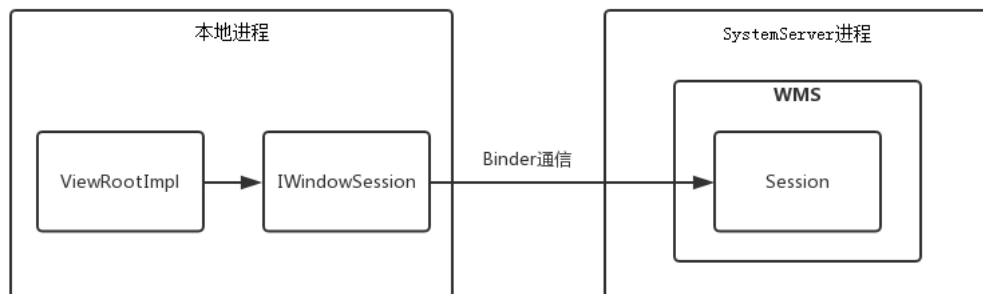


图 7-4 ViewRootImpl 与 WMS 通信

从图 7-4 可以看出,本地进程的 ViewRootImpl 要想和 WMS 进行通信需要经过 Session, 那么 Session 为何包含在 WMS 中呢? 我们接着往下看 Session 的 addToDisplay 方法, 如下所示:

```
frameworks/base/services/core/java/com/android/server/wm/Session.java
```

```

@Override
public int addToDisplay(IWindow window, int seq, WindowManager.LayoutParams attrs,
    int viewVisibility, int displayId, Rect outContentInsets, Rect outStableInsets,
    Rect outOutsets, InputChannel outInputChannel) {
    return mService.addWindow(this, window, seq, attrs, viewVisibility, displayId,
        outContentInsets, outStableInsets, outOutsets, outInputChannel);
}
  
```

在 addToDisplay 方法中调用了 WMS 的 addWindow 方法, 并将自身也就是 Session 作为参数传了进去, 每个应用程序进程都会对应一个 Session, WMS 会用 ArrayList 来保存这些 Session, 这就是为什么图 7-4 中的 WMS 包含 Session 的原因。这样剩下的工作就交给 WMS 来处理, 在 WMS 中会为此添加的窗口分配 Surface, 并确定窗口显示次序, 可见负责显示界面的是画布 Surface, 而不是窗口本身。WMS 会将它所管理的 Surface 交由 SurfaceFlinger 处理, SurfaceFlinger 会将这些 Surface 混合并绘制到屏幕上。

窗口添加的 WMS 处理部分会在第 8 章进行讲解, 系统窗口 StatusBar 的添加过程的时序图如图 7-5 所示。

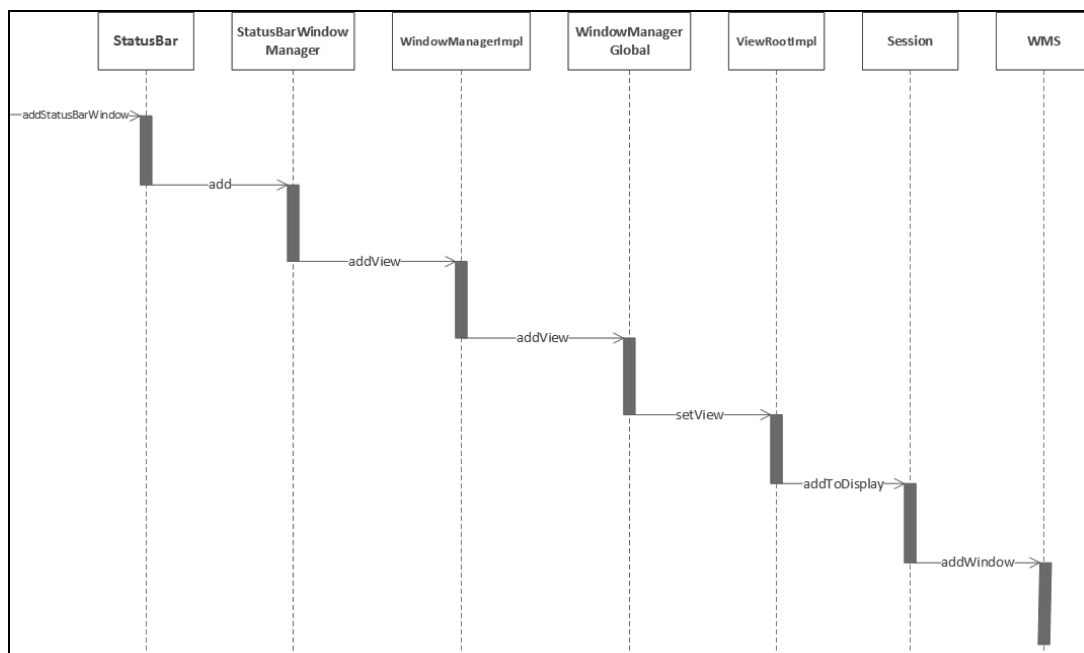


图 7-5 系统窗口 StatusBar 的添加过程的时序图

7.4.2 Activity 的添加过程

无论是哪种窗口，它的添加过程在 WMS 处理部分中基本是类似的，只不过会在权限和窗口显示次序等方面会有些不同。但是在 WindowManager 处理部分会有所不同，这里以最典型的应用程序窗口 Activity 为例，Activity 在启动过程中，如果 Activity 所在的进程不存在则会创建新的进程，创建新的进程之后就会运行代表主线程的实例 ActivityThread，ActivityThread 管理着当前应用程序进程的线程，这在 Activity 的启动过程中运用得很明显，当界面要与用户进行交互时，会调用 ActivityThread 的 handleResumeActivity 方法，如下所示：

frameworks/base/core/java/android/app/ActivityThread.java

```

final void handleResumeActivity(IBinder token,
    boolean clearHide, boolean isForward, boolean reallyResume, int seq, String reason)
{
    ...
    r = performResumeActivity(token, clearHide, reason); //1
    ...
    if (r.window == null && !a.mFinished && willBeVisible) {
        r.window = r.activity.getWindow();
    }
}

```



```

View decor = r.window.getDecorView();
decor.setVisibility(View.INVISIBLE);
ViewManager wm = a.getWindowManager();//2
WindowManager.LayoutParams l = r.window.getAttributes();
a.mDecor = decor;
l.type = WindowManager.LayoutParams.TYPE_BASE_APPLICATION;
l.softInputMode |= forwardBit;
if (r.mPreserveWindow) {
    a.mWindowAdded = true;
    r.mPreserveWindow = false;
    ViewRootImpl impl = decor.getViewRootImpl();
    if (impl != null) {
        impl.notifyChildRebuilt();
    }
}
if (a.mVisibleFromClient) {
    if (!a.mWindowAdded) {
        a.mWindowAdded = true;
        wm.addView(decor, l);//3
    } else {
        a.onWindowAttributesChanged(l);
    }
}
...
}

```

注释 1 处的 `performResumeActivity` 方法最终会调用 `Activity` 的 `onResume` 方法。在注释 2 处得到 `ViewManager` 类型的 `wm` 对象，在注释 3 处调用了 `ViewManager` 的 `addView` 方法，而 `addView` 方法则是在 `WindowManagerImpl` 中实现的，此后的过程在上面的系统窗口 `StatusBar` 的添加过程中已经讲过，唯一需要注意的是 `ViewManager` 的 `addView` 方法的第一个参数为 `DecorView`，这说明 `Acitivity` 窗口中会包含 `DecorView`。

7.4.3 Window的更新过程

Window 的更新过程和 Window 的添加过程是类似的。需要调用 `ViewManager` 的 `updateViewLayout` 方法，`updateViewLayout` 方法在 `WindowManagerImpl` 中实现，`WindowManagerImpl` 的 `updateViewLayout` 方法会调用 `WindowManagerGlobal` 的 `updateViewLayout` 方法，如下所示：

```
frameworks/base/core/java/android/view/WindowManagerGlobal.java
```

```
public void updateViewLayout(View view, ViewGroup.LayoutParams params) {
```

```

        if (view == null) {
            throw new IllegalArgumentException("view must not be null");
        }
        if (!(params instanceof WindowManager.LayoutParams)) {
            throw new IllegalArgumentException("Params must be WindowManager.
                LayoutParams");
        }
        final WindowManager.LayoutParams wparams = (WindowManager.LayoutParams)
            params;
        view.setLayoutParams(wparams); //1
        synchronized (mLock) {
            int index = findViewLocked(view, true); //2
            ViewRootImpl root = mRoots.get(index); //3
            mParams.remove(index); //4
            mParams.add(index, wparams); //5
            root.setLayoutParams(wparams, false); //6
        }
    }
}

```

注释 1 处将更新的参数设置到 View 中。注释 2 处得到要更新的窗口在 View 列表中的索引，注释 3 处在 ViewRootImpl 列表中根据索引得到窗口的 ViewRootImpl，注释 4 和注释 5 处用于更新布局参数列表，注释 6 处调用 ViewRootImpl 的 setLayoutParams 方法将更新的参数设置到 ViewRootImpl 中。ViewRootImpl 的 setLayoutParams 方法在最后会调用 ViewRootImpl 的 scheduleTraversals 方法，如下所示：

frameworks/base/core/java/android/view/ViewRootImpl.java

```

void scheduleTraversals() {
    if (!mTraversalScheduled) {
        mTraversalScheduled = true;
        mTraversalBarrier = mHandler.getLooper().getQueue().postSyncBarrier();
        mChoreographer.postCallback( Choreographer.CALLBACK_TRAVERSAL,
mTraversalRunnable, null); //1
        if (!mUnbufferedInputDispatch) {
            scheduleConsumeBatchedInput();
        }
        notifyRendererOfFramePending();
        pokeDrawLockIfNeeded();
    }
}

```

注释 1 处的 Choreographer 译为“舞蹈指导”，用于接收显示系统的 VSync 信号，在下一个帧渲染时控制执行一些操作。Choreographer 的 postCallback 方法用于发起添加回调，这个添加的回调将在下一帧被渲染时执行。这个添加的回调指的是注释 1 处的

TraversalRunnable 类型的 mTraversalRunnable，如下所示：

frameworks/base/core/java/android/view/ViewRootImpl.java

```
final class TraversalRunnable implements Runnable {
    @Override
    public void run() {
        doTraversal();
    }
}
```

在 TraversalRunnable 的 run 方法中调用了 doTraversal 方法，如下所示：

frameworks/base/core/java/android/view/ViewRootImpl.java

```
void doTraversal() {
    if (mTraversalScheduled) {
        mTraversalScheduled = false;
        mHandler.getLooper().getQueue().removeSyncBarrier(mTraversalBarrier);
        if (mProfile) {
            Debug.startMethodTracing("ViewAncestor");
        }
        performTraversals();
        if (mProfile) {
            Debug.stopMethodTracing();
            mProfile = false;
        }
    }
}
```

在 doTraversal 方法中又调用了 performTraversals 方法，performTraversals 方法使得 ViewTree 开始 View 的工作流程，如下所示：

在 frameworks/base/core/java/android/view/ViewRootImpl.java

```
private void performTraversals() {
    ...
    relayoutResult = relayoutWindow(params, viewVisibility, insetsPending);//1
    ...
    if (!mStopped) {
        int childWidthMeasureSpec = getRootMeasureSpec(mWidth, lp.width);
        int childHeightMeasureSpec = getRootMeasureSpec(mHeight, lp.height);
        performMeasure(childWidthMeasureSpec, childHeightMeasureSpec);//2
    }
}
if (didLayout) {
    performLayout(lp, desiredWindowWidth, desiredWindowHeight);//3
}
```

```

    ...
}
if (!cancelDraw && !newSurface) {
    if (!skipDraw || mReportNextDraw) {
        if (mPendingTransitions != null && mPendingTransitions.size() > 0) {
            for (int i = 0; i < mPendingTransitions.size(); ++i) {
                mPendingTransitions.get(i).startChangingAnimations();
            }
            mPendingTransitions.clear();
        }
        performDraw();//4
    }
}
...
}

```

注释 1 处的 `relayoutWindow` 方法内部会调用 `IWindowSession` 的 `relayout` 方法来更新 Window 视图，这一过程和 7.4.1 节中图 7-4 的原理是一样的，最终会调用 WMS 的 `relayoutWindow` 方法。除此之外，`performTraversals` 方法还会在注释 2、3、4 处分别调用 `performMeasure`、`performLayout` 和 `performDraw` 方法，它们的内部又会调用 View 的 `measure`、`layout` 和 `draw` 方法，这样就完成了 View 的工作流程。在 `performTraversals` 方法中更新了 Window 视图，又执行 Window 中的 View 的工作流程，这样就完成了 Window 的更新，至于 Window 的删除过程将在第 8 章进行讲解。

7.5 本章小结

本章我们学习了 `WindowManager` 的相关知识点，包括 `WindowManager` 的关联类、Window 的属性和 Window 的操作，在这一章我们知道了 Window 的属性，这对应用开发会有所帮助，了解了 Window、`WindowManager` 和 WMS 之间的关系，也知道了 Window 的操作分为两大部分并且最终都是由 WMS 处理的，这为第 8 章讲解 WMS 做好了铺垫。

第 8 章

理解 WindowManagerService

关联章节：第 7 章 理解 WindowManager

在第 7 章我们学习了 WindowManager，本章将接着介绍 WindowManager 的管理者 WMS，WMS 不只是 WindowManager 的管理者，它还有很多重要的职责，本章将介绍 WMS 的职责、WMS 的创建过程、WMS 的重要成员以及 Window 的添加和删除过程，阅读本章前请先阅读第 7 章的内容。

8.1 WMS 的职责

很多读者都知道 WMS 是 Android 中重要的服务，它是 WindowManager 的管理者，WMS 无论对于应用开发还是 Framework 开发都是重要的知识点，究其原因是因为 WMS 有很多职责，每个职责都会涉及重要且复杂的系统，这使得 WMS 就像一个十字路口的交通灯一样，没有了这个交通灯，十字路口就无法正常通车。WMS 的职责用简洁的语言介绍主要有以下几点。

1. 窗口管理

WMS 是窗口的管理者，它负责窗口的启动、添加和删除，另外窗口的大小和层级也是由 WMS 进行管理的。窗口管理的核心成员有 DisplayContent、WindowToken 和 WindowState。

2. 窗口动画

窗口间进行切换时，使用窗口动画可以显得更炫一些，窗口动画由 WMS 的动画子系统来负责，动画子系统的管理者为 WindowAnimator。

3. 输入系统的中转站

通过对窗口的触摸从而产生触摸事件，InputManagerService (IMS) 会对触摸事件进行处理，它会寻找一个最合适的窗口来处理触摸反馈信息，WMS 是窗口的管理者，它作为输入系统的中转站再合适不过了。

4. Surface 管理

窗口并不具备绘制的功能，因此每个窗口都需要有一块 Surface 来供自己绘制，为每个窗口分配 Surface 是由 WMS 来完成的。

WMS 的职责可以简单总结为如图 8-1 所示。

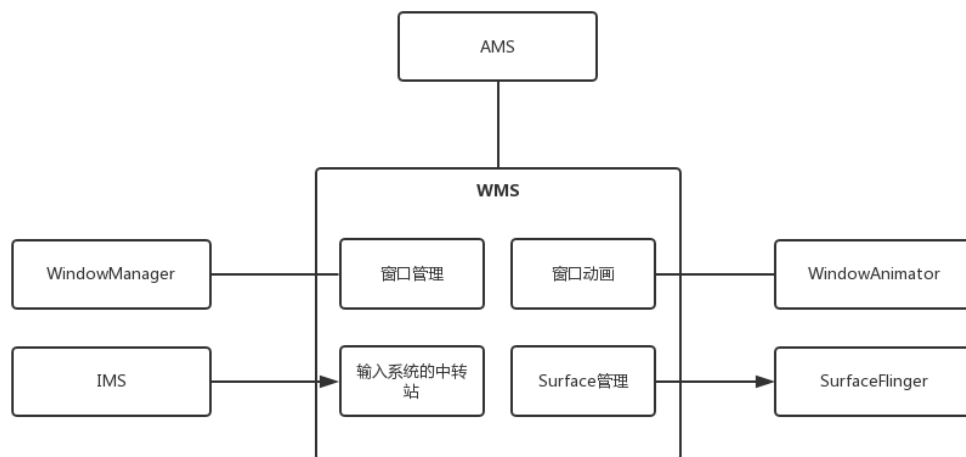


图 8-1 WMS 的职责

从 WMS 的职责可以看出 WMS 很复杂，与它关联的有窗口管理、窗口动画、输入系统和 Surface，它们每一个都是重要且复杂的系统，本章只介绍其中的窗口管理，因为它和应用开发的关联比较紧密。

8.2 WMS的创建过程

WMS 涉及的知识点非常多,但是无论这些知识点如何多,我们还是十分有必要先知道 WMS 是如何创建的。WMS 是在 SystemServer 进程中创建的,不了解 SystemServer 进程的可以查看 2.3 节的内容,先来查看 SystemServer 的 main 方法,如下所示:

```
frameworks/base/services/java/com/android/server/SystemServer.java
```

```
public static void main(String[] args) {
    new SystemServer().run();
}
```

在 main 方法中只调用了 SystemServer 的 run 方法,如下所示:

```
frameworks/base/services/java/com/android/server/SystemServer.java
```

```
private void run() {
    try {
        ...
        //创建消息 Looper
        Looper.prepareMainLooper();
        //加载了动态库 libandroid_servers.so
        System.loadLibrary("android_servers");//1
        performPendingShutdown();
        // 创建系统的 Context
        createSystemContext();
        // 创建 SystemServiceManager
        mSystemServiceManager = new SystemServiceManager(mSystemContext);//2
        mSystemServiceManager.setRuntimeRestarted(mRuntimeRestart);
        LocalServices.addService(SystemServiceManager.class, mSystemServiceManager);
        SystemServerInitThreadPool.get();
    } finally {
        traceEnd(); // InitBeforeStartServices
    }
    try {
        traceBeginAndSlog("StartServices");
        //启动引导服务
        startBootstrapServices();//3
        //启动核心服务
        startCoreServices();//4
        //启动其他服务
        startOtherServices();//5
        SystemServerInitThreadPool.shutdown();
    } catch (Throwable ex) {
```

```

        Slog.e("System", "*****");
        Slog.e("System", "***** Failure starting system services", ex);
        throw ex;
    } finally {
        traceEnd();
    }
    ...
}

```

在注释 1 处加载了动态库 libandroid_servers.so。接下来在注释 2 处创建 SystemServiceManager，它会对系统的服务进行创建、启动和生命周期管理。在注释 3 处的 startBootstrapServices 方法中用 SystemServiceManager 启动了 ActivityManagerService、PowerManagerService、PackageManagerService 等服务。在注释 4 处的 startCoreServices 方法中则启动了 DropBoxManagerService、BatteryService、UsageStatsService 和 WebViewUpdateService。在注释 5 处的 startOtherServices 方法中启动了 CameraService、AlarmManagerService、VrManagerService 等服务。这些服务的父类均为 SystemService。从注释 3、注释 4、注释 5 处的方法可以看出，官方把系统服务分为了三种类型，分别是引导服务、核心服务和其他服务，其中其他服务是一些非紧要和不需要立即启动的服务，WMS 就是其他服务的一种。我们来查看在 startOtherServices 方法中是如何启动 WMS 的：

frameworks/base/services/java/com/android/server/SystemServer.java

```

private void startOtherServices() {
    ...
    traceBeginAndSlog("InitWatchdog");
    final Watchdog watchdog = Watchdog.getInstance();//1
    watchdog.init(context, mActivityManagerService);//2
    traceEnd();
    traceBeginAndSlog("StartInputManagerService");
    inputManager = new InputManagerService(context);//3
    traceEnd();
    traceBeginAndSlog("StartWindowManagerService");
    ConcurrentUtils.waitForFutureNoInterrupt(mSensorServiceStart, START_
    SENSOR_SERVICE);
    mSensorServiceStart = null;
    wm = WindowManagerService.main(context, inputManager,
        mFactoryTestMode != FactoryTest.FACTORY_TEST_LOW_LEVEL,
        !mFirstBoot, mOnlyCore, new PhoneWindowManager());//4
    ServiceManager.addService(Context.WINDOW_SERVICE, wm);//5
    ServiceManager.addService(Context.INPUT_SERVICE, inputManager);//6
    traceEnd();
    ...
    try {

```



```

        wm.displayReady();//7
    } catch (Throwable e) {
        reportWtf("making display ready", e);
    }
    ...
try {
    wm.systemReady();//8
    } catch (Throwable e) {
        reportWtf("making Window Manager Service ready", e);
    }
    ...
}

```

startOtherServices 方法用于启动其他服务，其他服务大概有 100 个左右，上面的代码只列出了 WMS 以及和它相关的 IMS 的启动逻辑，剩余的其他服务的启动逻辑也都大同小异。在注释 1、注释 2 处分别得到 Watchdog 实例并对它进行初始化，Watchdog 用来监控系统的一些关键服务的运行状况，后文会再次提到它。在注释 3 处创建了 IMS，并赋值给 IMS 类型的 inputManager 对象。在注释 4 处执行了 WMS 的 main 方法，其内部会创建 WMS，需要注意的是 main 方法其中一个传入的参数就是在注释 1 处创建的 IMS，WMS 是输入事件的中转站，其内部包含了 IMS 引用并不意外。结合上文，我们可以得知 WMS 的 main 方法是运行在 SystemServer 的 run 方法中的，换句话说就是运行在“system_server”线程中，后面会再次提到“system_server”线程。在注释 5 和注释 6 处分别将 WMS 和 IMS 注册到 ServiceManager 中，这样如果某个客户端想要使用 WMS，就需要先去 ServiceManager 中查询信息，然后根据信息与 WMS 所在的进程建立通信通路，客户端就可以使用 WMS 了。在注释 7 处用来初始化屏幕显示信息，在注释 8 处则用来通知 WMS，系统的初始化工作已经完成，其内部调用了 WindowManagerPolicy 的 systemReady 方法。我们来查看注释 4 处 WMS 的 main 方法，如下所示：

```
frameworks/base/services/core/java/com/android/server/wm/WindowManagerService.java
```

```

public static WindowManagerService main(final Context context, final InputManager
Service im, final boolean haveInputMethods, final boolean showBootMsgs, final
boolean onlyCore, WindowManagerPolicy policy) {
    DisplayThread.getHandler().runWithScissors(() ->{
        sInstance = new WindowManagerService(context, im, haveInputMethods,
        showBootMsgs, onlyCore, policy), 0);
        return sInstance;
    })
}

```

在注释 1 处调用了 DisplayThread 的 getHandler 方法，用来得到 DisplayThread 的 Handler

实例。DisplayThread 是一个单例的前台线程，这个线程用来处理需要低延时显示的相关操作，并只能由 WindowManager、DisplayManager 和 InputManager 实时执行快速操作。在注释 1 处的 runWithScissors 方法中使用了 Java 8 中的 Lambda 表达式，它等价于如下代码：

```
DisplayThread.getHandler().runWithScissors(new Runnable() {
    @Override
    public void run() {
        sInstance = new WindowManagerService(context, im, haveInputMethods,
            showBootMsgs, onlyCore, policy);//2
    }
}, 0);
```

在注释 2 处创建了 WMS 的实例，这个过程运行在 Runnable 的 run 方法中，而 Runnable 则传到了 DisplayThread 对应 Handler 的 runWithScissors 方法中，说明 WMS 的创建是运行在 android.display 线程中的。需要注意的是，runWithScissors 方法的第二个参数传入的是 0，后面会提到。下面来查看 Handler 的 runWithScissors 方法做了什么：

frameworks/base/core/java/android/os/Handler.java

```
public final boolean runWithScissors(final Runnable r, long timeout) {
    if (r == null) {
        throw new IllegalArgumentException("runnable must not be null");
    }
    if (timeout < 0) {
        throw new IllegalArgumentException("timeout must be non-negative");
    }
    if (Looper.myLooper() == mLooper) {//1
        r.run();
        return true;
    }
    BlockingRunnable br = new BlockingRunnable(r);
    return br.postAndWait(this, timeout);
}
```

开头对传入的 Runnable 和 timeout 进行了判断，如果 Runnable 为 null 或者 timeout 小于 0 则抛出异常。在注释 1 处根据每个线程只有一个 Looper 的原理来判断当前的线程（system_server 线程）是否是 Handler 所指向的线程（android.display 线程），如果是则直接执行 Runnable 的 run 方法，如果不是则调用 BlockingRunnable 的 postAndWait 方法，并将当前线程的 Runnable 作为参数传进去，BlockingRunnable 是 Handler 的内部类，代码如下所示：

```
frameworks/base/core/java/android/os/Handler.java
```

```
private static final class BlockingRunnable implements Runnable {
    private final Runnable mTask;
    private boolean mDone;
    public BlockingRunnable(Runnable task) {
        mTask = task;
    }
    @Override
    public void run() {
        try {
            mTask.run();//1
        } finally {
            synchronized (this) {
                mDone = true;
                notifyAll();
            }
        }
    }
}

public boolean postAndWait(Handler handler, long timeout) {
    if (!handler.post(this)) { //2
        return false;
    }
    synchronized (this) {
        if (timeout > 0) {
            final long expirationTime = SystemClock.uptimeMillis() + timeout;
            while (!mDone) {
                long delay = expirationTime - SystemClock.uptimeMillis();
                if (delay <= 0) {
                    return false; // timeout
                }
                try {
                    wait(delay);
                } catch (InterruptedException ex) {
                }
            }
        } else {
            while (!mDone) {
                try {
                    wait();//3
                } catch (InterruptedException ex) {
                }
            }
        }
    }
}
```

```

    }
    return true;
}
}

```

在注释 2 处将当前的 `BlockingRunnable` 添加到 `Handler` 的任务队列中。前面 `runWithScissors` 方法的第二个参数为 0，因此 `timeout` 等于 0，这样如果 `mDone` 为 `false` 的话会一直调用注释 3 处的 `wait` 方法使得当前线程（`system_server` 线程）进入等待状态，那么等待的是哪个线程呢？我们往上看，在注释 1 处执行了传入的 `Runnable` 的 `run` 方法（运行在 `android.display` 线程），执行完毕后在 `finally` 代码块中将 `mDone` 设置为 `true`，并调用 `notifyAll` 方法唤醒处于等待状态的线程，这样就不会继续调用注释 3 处的 `wait` 方法。因此得出结论，`system_server` 线程等待的就是 `android.display` 线程，一直到 `android.display` 线程执行完毕再执行 `system_server` 线程，这是因为 `android.display` 线程内部执行了 `WMS` 的创建，而 `WMS` 的创建优先级要更高。`WMS` 的创建就讲到这里，最后查看 `WMS` 的构造方法：

`frameworks/base/services/core/java/com/android/server/wm/WindowManagerService.java`

```

private WindowManagerService(Context context, InputManagerService inputManager,
    boolean haveInputMethods, boolean showBootMsgs, boolean onlyCore,
    WindowManagerPolicy policy) {
    ...
    mInputManager = inputManager;//1
    ...
    mDisplayManager = (DisplayManager)context.getSystemService(Context.DISPLAY_
        SERVICE);
    mDisplays = mDisplayManager.getDisplays();//2
    for (Display display : mDisplays) {
        createDisplayContentLocked(display);//3
    }
    ...
    mActivityManager = ActivityManager.getService();//4
    ...
    mAnimator = new WindowAnimator(this);//5
    mAllowTheaterModeWakeFromLayout = context.getResources().getBoolean(
        com.android.internal.R.bool.config_allowTheaterModeWakeFromWindow
        Layout);
    LocalServices.addService(WindowManagerInternal.class, new LocalService());
    initPolicy();//6
    Watchdog.getInstance().addMonitor(this);//7
    ...
}

```

注释 1 处用来保存传进来的 IMS，这样 WMS 就持有了 IMS 的引用。在注释 2 处通过 DisplayManager 的 getDisplays 方法得到 Display 数组（每个显示设备都有一个 Display 实例），接着遍历 Display 数组，在注释 3 处的 createDisplayContentLocked 方法将 Display 封装成 DisplayContent，DisplayContent 用来描述一块屏幕。在注释 4 处得到 AMS 实例，并赋值给 mActivityManager，这样 WMS 就持有了 AMS 的引用。在注释 5 处创建了 WindowAnimator，它用于管理所有的窗口动画。在注释 6 处初始化了窗口管理策略的接口类 WindowManagerPolicy（WMP），它用来定义一个窗口策略所要遵循的通用规范。在注释 7 处将自身也就是 WMS 通过 addMonitor 方法添加到 Watchdog 中，Watchdog 用来监控系统的一些关键服务的运行状况（比如传入的 WMS 的运行状况），这些被监控的服务都会实现 Watchdog.Monitor 接口。Watchdog 每分钟都会对被监控的系统服务进行检查，如果被监控的系统服务出现了死锁，则会杀死 Watchdog 所在的进程，也就是 SystemServer 进程。查看注释 6 处的 initPolicy 方法，如下所示：

```
frameworks/base/services/core/java/com/android/server/wm/WindowManagerService.java
```

```
private void initPolicy() {
   UiThread.getHandler().runWithScissors(new Runnable() {
        @Override
        public void run() {
            WindowManagerPolicyThread.set(Thread.currentThread(), Looper.myLooper());
            mPolicy.init(mContext, WindowManagerService.this, WindowManager
                Service.this);//1
        }
    }, 0);
}
```

initPolicy 方法和此前讲的 WMS 的 main 方法的实现类似，在注释 1 处执行了 WMP 的 init 方法，WMP 是一个接口，init 方法具体在 PhoneWindowManager（PWM）中实现。PWM 的 init 方法运行在 android.ui 线程中，它的优先级要高于 initPolicy 方法所在的 android.display 线程，因此 android.display 线程要等 PWM 的 init 方法执行完毕后，处于等待状态的 android.display 线程才会被唤醒从而继续执行下面的代码。本文共提到了 3 个线程，分别是 system_server、android.display 和 android.ui，为了便于理解，下面给出这 3 个线程之间的关系，如图 8-2 所示。

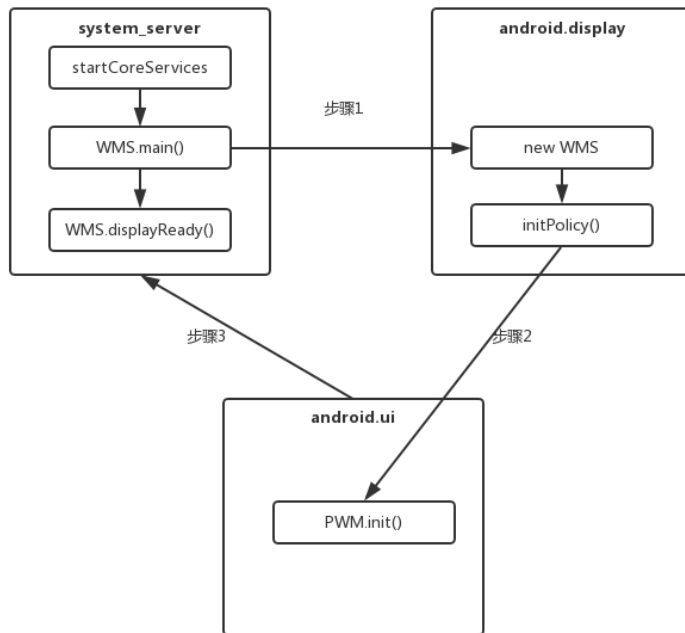


图 8-2 三个线程之间的关系

从图 8-2 可以看出，三个线程之间的关系分为三个步骤来实现：

(1) 首先在 `system_server` 线程中执行了 `SystemService` 的 `startOtherServices` 方法，在 `startOtherServices` 方法中会调用 `WMS` 的 `main` 方法，`main` 方法会创建 `WMS`，创建的过程在 `android.display` 线程中实现，创建 `WMS` 的优先级更高，因此 `system_server` 线程要等 `WMS` 创建完成后，处于等待状态的 `system_server` 线程才会被唤醒从而继续执行下面的代码。

(2) 在 `WMS` 的构造方法中会调用 `WMS` 的 `initPolicy` 方法，在 `initPolicy` 方法中又会调用 `PWM` 的 `init` 方法，`PWM` 的 `init` 方法在 `android.ui` 线程中运行，它的优先级要高于 `android.display` 线程，因此“`android.display`”线程要等 `PWM` 的 `init` 方法执行完毕后，处于等待状态的 `android.display` 线程才会被唤醒从而继续执行下面的代码。

(3) `PWM` 的 `init` 方法执行完毕后，`android.display` 线程就完成了 `WMS` 的创建，等待的 `system_server` 线程被唤醒后继续执行 `WMS` 的 `main` 方法后的代码逻辑，比如 `WMS` 的 `displayReady` 方法用来初始化屏幕显示信息（`SystemService` 的 `startOtherServices` 方法的注释 7 处）。

8.3 WMS的重要成员

要想更好地理解 WMS, 不但要了解 WMS 是如何创建的, 还要知道 WMS 的重要成员, 这里的重要成员指的是 WMS 的部分成员变量, 如下所示:

```
frameworks/base/services/core/java/com/android/server/wm/WindowManagerService.java
```

```
final WindowManagerPolicy mPolicy;
final IActivityManager mActivityManager;
final ActivityManagerInternal mAmInternal;
final AppOpsManager mAppOps;
final DisplaySettings mDisplaySettings;
...
final ArraySet<Session> mSessions = new ArraySet<>();
final WindowHashMap mWindowMap = new WindowHashMap();
final ArrayList<AppWindowToken> mFinishedStarting = new ArrayList<>();
final ArrayList<AppWindowToken> mFinishedEarlyAnim = new ArrayList<>();
final ArrayList<AppWindowToken> mWindowReplacementTimeouts = new ArrayList<>();
final ArrayList<WindowState> mResizingWindows = new ArrayList<>();
final ArrayList<WindowState> mPendingRemove = new ArrayList<>();
WindowState[] mPendingRemoveTmp = new WindowState[20];
final ArrayList<WindowState> mDestroySurface = new ArrayList<>();
final ArrayList<WindowState> mDestroyPreservedSurface = new ArrayList<>();
...
final H mH = new H();
...
final WindowAnimator mAnimator;
...
final InputManagerService mInputManager
```

上面的代码列出了 WMS 的部分成员变量, 下面分别对它们进行简单介绍。

1. mPolicy: WindowManagerPolicy

mPolicy 是 WindowManagerPolicy (WMP) 类型的变量。WindowManagerPolicy 是窗口管理策略的接口类, 用来定义一个窗口策略所要遵循的通用规范, 并提供了 WindowManager 所有的特定的 UI 行为。它的具体实现类为 PhoneWindowManager, 这个实现类在 WMS 创建时被创建。WMP 允许定制窗口层级和特殊窗口类型以及关键的调度和布局。

2. mSessions: ArraySet

mSessions 是 ArraySet 类型的变量, 元素类型为 Session, 它主要用于进程间通信, 其他的应用程序进程想要和 WMS 进程进行通信就需要经过 Session, 并且每个应用程序进程

都会对应一个 Session, WMS 保存这些 Session 用来记录所有向 WMS 提出窗口管理服务的客户端。

3. mWindowMap: WindowHashMap

mWindowMap 是 WindowHashMap 类型的变量, WindowHashMap 继承了 HashMap, 它限制了 HashMap 的 key 值的类型为 IBinder, value 值的类型为 WindowState。WindowState 用于保存窗口的信息, 在 WMS 中它用来描述一个窗口。综上所述, mWindowMap 就是用来保存 WMS 中各种窗口的集合。

4. mFinishedStarting: ArrayList

mFinishedStarting 是 ArrayList 类型的变量, 元素类型为 AppWindowToken, 它是 WindowToken 的子类。要想理解 mFinishedStarting 的含义, 需要先了解 WindowToken 是什么。WindowToken 主要有两个作用:

- 可以理解为窗口令牌, 当应用程序想要向 WMS 申请新创建一个窗口, 则需要向 WMS 出示有效的 WindowToken。AppWindowToken 作为 WindowToken 的子类, 主要用来描述应用程序的 WindowToken 结构, 应用程序中每个 Activity 都对应一个 AppWindowToken。
- WindowToken 会将同一个组件 (比如同一个 Activity) 的窗口 (WindowState) 集合在一起, 方便管理。

mFinishedStarting 就是用于存储已经完成启动的应用程序窗口 (比如 Activity) 的 AppWindowToken 的列表。除了 mFinishedStarting 外, 还有类似的 mFinishedEarlyAnim 和 mWindowReplacementTimeouts, 其中 mFinishedEarlyAnim 存储了已经完成窗口绘制并且不需要展示任何已保存 surface 的应用程序窗口的 AppWindowToken。mWindowReplacementTimeout 存储了等待更换的应用程序窗口的 AppWindowToken, 如果更换不及时, 旧窗口就需要被处理。

5. mResizingWindows: ArrayList

mResizingWindows 是 ArrayList 类型的变量, 元素类型为 WindowState。mResizingWindows 是用来存储正在调整大小的窗口的列表。与 mResizingWindows 类似的还有 mPendingRemove、mDestroySurface 和 mDestroyPreservedSurface 等, 其中 mPendingRemove 是在内存耗尽时设置的, 里面存有需要强制删除的窗口, mDestroySurface 里面存有需要被销毁的 Surface, mDestroyPreservedSurface 里面存有窗口需要保存的等待销毁的 Surface,

为什么窗口要保存这些 Surface? 这是因为当窗口经历 Surface 变化时, 窗口需要一直保持旧 Surface, 直到新 Surface 的第一帧绘制完成。

6. mAnimator: WindowAnimator

mAnimator 是 WindowAnimator 类型的变量, 用于管理窗口的动画以及特效动画。

7. mHandler: Handler

mHandler 是 Handler 类型的变量, 系统的 Handler 类, 用于将任务加入到主线程的消息队列中, 这样代码逻辑就会在主线程中执行。

8. mInputManager: InputManagerService

mInputManager 是 InputManagerService 类型的变量, 输入系统的管理者。InputManagerService (IMS) 会对触摸事件进行处理, 它会寻找一个最合适的窗口来处理触摸反馈信息, WMS 是窗口的管理者, 因此 WMS 作为输入系统的中转站是再合适不过了。

8.4 Window的添加过程 (WMS处理部分)

我们知道 Window 的操作分为两大部分, 一部分是 WindowManager 处理部分, 另一部分是 WMS 处理部分, 在 7.4.1 节中我们学习了 Window 添加过程的 WindowManager 处理部分, 这一节我们接着来学习 Window 的添加过程的 WMS 处理部分。无论是系统窗口还是 Activity, 它们的 Window 的添加过程都会调用 WMS 的 addWindow 方法, 由于这个方法代码逻辑比较多, 这里分为 3 个部分来阅读。

```
frameworks/base/services/core/java/com/android/server/wm/WindowManagerService.java
```

1. addWindow 方法 part1

```
public int addWindow(Session session, IWindow client, int seq,
    WindowManager.LayoutParams attrs, int viewVisibility, int displayId,
    Rect outContentInsets, Rect outStableInsets, Rect outOutsets,
    InputChannel outInputChannel) {
    int[] appOp = new int[1];
    int res = mPolicy.checkAddPermission(attrs, appOp); //1
    if (res != WindowManagerGlobal.ADD_OKAY) {
        return res;
    }
}
```

```

...
synchronized(mWindowMap) {
    if (!mDisplayReady) {
        throw new IllegalStateException("Display has not been initialized");
    }
    final DisplayContent displayContent = mRoot.getDisplayContentOrCreate
    (displayId);//2
    if (displayContent == null) {
        Slog.w(TAG_WM, "Attempted to add window to a display that does not exist: "
            + displayId + ". Aborting.");
        return WindowManagerGlobal.ADD_INVALID_DISPLAY;
    }
    ...
    if (type >= FIRST_SUB_WINDOW && type <= LAST_SUB_WINDOW) {//3
    parentWindow = windowForClientLocked(null, attrs.token, false);//4
    if (parentWindow == null) {
        Slog.w(TAG_WM, "Attempted to add window with token that is not a
            window: " + attrs.token + ". Aborting.");
        return WindowManagerGlobal.ADD_BAD_SUBWINDOW_TOKEN;
    }
    if (parentWindow.mAttrs.type >= FIRST_SUB_WINDOW
        && parentWindow.mAttrs.type <= LAST_SUB_WINDOW) {
        Slog.w(TAG_WM, "Attempted to add window with token that is a
            sub-window: " + attrs.token + ". Aborting.");
        return WindowManagerGlobal.ADD_BAD_SUBWINDOW_TOKEN;
    }
    }
    ...
}
...
}

```

WMS 的 addWindow 返回的是 addWindow 的各种状态，比如添加 Window 成功，无效的 display 等，这些状态被定义在 WindowManagerGlobal 中。在注释 1 处根据 Window 的属性，调用 WMP 的 checkAddPermission 方法来检查权限，具体在 PhoneWindowManager 的 checkAddPermission 方法中实现，如果没有权限则不会执行后续的代码逻辑。在注释 2 处通过 displayId 来获得窗口要添加到哪个 DisplayContent 上，如果没有找到 DisplayContent，则返回 WindowManagerGlobal.ADD_INVALID_DISPLAY 这一状态，其中 DisplayContent 用来描述一块屏幕。在注释 3 处，type 代表一个窗口的类型，它的数值介于 FIRST_SUB_WINDOW 和 LAST_SUB_WINDOW 之间 (1000~1999)，这个数值定义在 WindowManager 中，说明这个窗口是一个子窗口。在注释 4 处，attrs.token 是 IBinder 类

型的对象，`windowForClientLocked` 方法内部会根据 `attrs.token` 作为 key 值从 `mWindowMap` 中得到该子窗口的父窗口。接着对父窗口进行判断，如果父窗口为 `null` 或者 `type` 的取值范围不正确则会返回错误的状态。

2. addWindow 方法 part2

```
...
AppWindowToken atoken = null;
final boolean hasParent = parentWindow != null;
WindowToken token = displayContent.getWindowToken(
    hasParent ? parentWindow.mAttrs.token : attrs.token); //1
final int rootType = hasParent ? parentWindow.mAttrs.type : type; //2
boolean addToastWindowRequiresToken = false;
if (token == null) {
    if (rootType >= FIRST_APPLICATION_WINDOW && rootType <= LAST_APPLICATION_
        WINDOW) {
        Slog.w(TAG_WM, "Attempted to add application window with unknown token "
            + attrs.token + ". Aborting.");
        return WindowManagerGlobal.ADD_BAD_APP_TOKEN;
    }
    if (rootType == TYPE_INPUT_METHOD) {
        Slog.w(TAG_WM, "Attempted to add input method window with unknown token "
            + attrs.token + ". Aborting.");
        return WindowManagerGlobal.ADD_BAD_APP_TOKEN;
    }
    if (rootType == TYPE_VOICE_INTERACTION) {
        Slog.w(TAG_WM, "Attempted to add voice interaction window with unknown
            token " + attrs.token + ". Aborting.");
        return WindowManagerGlobal.ADD_BAD_APP_TOKEN;
    }
    if (rootType == TYPE_WALLPAPER) {
        Slog.w(TAG_WM, "Attempted to add wallpaper window with unknown token "
            + attrs.token + ". Aborting.");
        return WindowManagerGlobal.ADD_BAD_APP_TOKEN;
    }
    ...
    if (type == TYPE_TOAST) {
        if (doesAddToastWindowRequireToken(attrs.packageName, callingUid,
            parentWindow)) {
            Slog.w(TAG_WM, "Attempted to add a toast window with unknown token "
                + attrs.token + ". Aborting.");
            return WindowManagerGlobal.ADD_BAD_APP_TOKEN;
        }
    }
}
```

```

        final IBinder binder = attrs.token != null ? attrs.token : client.asBinder();
        token = new WindowToken(this, binder, type, false, displayContent,
            session.mCanAddInternalSystemWindow); //3
    } else if (rootType >= FIRST_APPLICATION_WINDOW && rootType <= LAST_
        APPLICATION_WINDOW) { //4
        atoken = token.asAppWindowToken(); //5
        if (atoken == null) {
            Slog.w(TAG_WM, "Attempted to add window with non-application token "
                + token + ". Aborting.");
            return WindowManagerGlobal.ADD_NOT_APP_TOKEN;
        } else if (atoken.removed) {
            Slog.w(TAG_WM, "Attempted to add window with exiting application token "
                + token + ". Aborting.");
            return WindowManagerGlobal.ADD_APP_EXITING;
        }
    } else if (rootType == TYPE_INPUT_METHOD) {
        if (token.windowType != TYPE_INPUT_METHOD) {
            Slog.w(TAG_WM, "Attempted to add input method window with bad token "
                + attrs.token + ". Aborting.");
            return WindowManagerGlobal.ADD_BAD_APP_TOKEN;
        }
    }
}
...

```

在注释 1 处通过 `displayContent` 的 `getWindowToken` 方法得到 `WindowToken`。在注释 2 处，如果有父窗口就将父窗口的 `type` 值赋值给 `rootType`，如果没有将当前窗口的 `type` 值赋值给 `rootType`。接下来如果 `WindowToken` 为 `null`，则根据 `rootType` 或者 `type` 的值进行区分判断，如果 `rootType` 值等于 `TYPE_INPUT_METHOD`、`TYPE_WALLPAPER` 等值时，则返回状态值 `WindowManagerGlobal.ADD_BAD_APP_TOKEN`，说明 `rootType` 值等于 `TYPE_INPUT_METHOD`、`TYPE_WALLPAPER` 等值时是不允许 `WindowToken` 为 `null` 的。通过多次的条件判断筛选，最后会在注释 3 处隐式创建 `WindowToken`，这说明当我们添加窗口时可以不向 WMS 提供 `WindowToken`，前提是 `rootType` 和 `type` 的值不为前面条件判断筛选的值。`WindowToken` 隐式和显式的创建肯定是要加以区分的，注释 3 处的第 4 个参数为 `false` 就代表这个 `WindowToken` 是隐式创建的。接下来的代码逻辑就是 `WindowToken` 不为 `null` 的情况，根据 `rootType` 和 `type` 的值进行判断，比如在注释 4 处判断如果窗口为应用程序窗口，在注释 5 处将 `WindowToken` 转换为专门针对应用程序窗口的 `AppWindowToken`，然后根据 `AppWindowToken` 的值进行后续的判断。

3. addWindow 方法 part3

```

...
final WindowState win = new WindowState(this, session, client, token, parentWindow,
    appOp[0], seq, attrs, viewVisibility, session.mUid,
    session.mCanAddInternalSystemWindow); //1
if (win.mDeathRecipient == null) { //2
    Slog.w(TAG_WM, "Adding window client " + client.asBinder()
        + " that is dead, aborting.");
    return WindowManagerGlobal.ADD_APP_EXITING;
}
if (win.getDisplayContent() == null) { //3
    Slog.w(TAG_WM, "Adding window to Display that has been removed.");
    return WindowManagerGlobal.ADD_INVALID_DISPLAY;
}
mPolicy.adjustWindowParamsLw(win.mAttrs); //4
win.setShowToOwnerOnlyLocked(mPolicy.checkShowToOwnerOnly(attrs));
res = mPolicy.prepareAddWindowLw(win, attrs); //5
...
win.attach();
mWindowMap.put(client.asBinder(), win); //6
if (win.mAppOp != AppOpsManager.OP_NONE) {
    int startOpResult = mAppOps.startOpNoThrow(win.mAppOp, win.getOwningUid(),
        win.getOwningPackage());
    if ((startOpResult != AppOpsManager.MODE_ALLOWED) &&
        (startOpResult != AppOpsManager.MODE_DEFAULT)) {
        win.setAppOpVisibilityLw(false);
    }
}
final AppWindowToken aToken = token.asAppWindowToken();
if (type == TYPE_APPLICATION_STARTING && aToken != null) {
    aToken.startingWindow = win;
    if (DEBUG_STARTING_WINDOW) Slog.v(TAG_WM, "addWindow: " + aToken
        + " startingWindow=" + win);
}
boolean imMayMove = true;
win.mToken.addWindow(win); //7
if (type == TYPE_INPUT_METHOD) {
    win.mGivenInsetsPending = true;
    setInputMethodWindowLocked(win);
    imMayMove = false;
} else if (type == TYPE_INPUT_METHOD_DIALOG) {
    displayContent.computeImeTarget(true /* updateImeTarget */);
    imMayMove = false;
} else {

```

```

        if (type == TYPE_WALLPAPER) {
            displayContent.mWallpaperController.clearLastWallpaperTimeoutTime();
            displayContent.pendingLayoutChanges |= FINISH_LAYOUT_REDO_WALLPAPER;
        } else if ((attrs.flags & FLAG_SHOW_WALLPAPER) != 0) {
            displayContent.pendingLayoutChanges |= FINISH_LAYOUT_REDO_WALLPAPER;
        } else if (displayContent.mWallpaperController.isBelowWallpaperTarget(
            win)) {
            displayContent.pendingLayoutChanges |= FINISH_LAYOUT_REDO_WALLPAPER;
        }
    }
    ...

```

在注释 1 处创建了 WindowState，它存有窗口的所有的状态信息，在 WMS 中它代表一个窗口。在创建 WindowState 传入的参数中，this 指的是 WMS，client 指的是 IWindow，IWindow 会将 WMS 中窗口管理的操作回调给 ViewRootImpl，token 指的是 WindowToken 紧接着在注释 2 和注释 3 处分别判断请求添加窗口的客户端是否已经死亡、窗口的 DisplayContent 是否为 null，如果是则不会再执行下面的代码逻辑。在注释 4 处调用了 WMP 的 adjustWindowParamsLw 方法，该方法在 PhoneWindowManager 中实现，此方法会根据窗口的 type 对窗口的 LayoutParams 的一些成员变量进行修改。在注释 5 处调用 WMP 的 prepareAddWindowLw 方法，用于准备将窗口添加到系统中。在注释 6 处将 WindowState 添加到 mWindowMap 中。在注释 7 处将 WindowState 添加到该 WindowState 对应的 WindowToken 中（实际是保存在 WindowToken 的父类 WindowContainer 中），这样 WindowToken 就包含了同一个组件的 WindowState。

4. addWindow 方法总结

addWindow 方法分了 3 个部分来进行讲解，主要就是做了下面 4 件事：

- 对所添加的窗口进行检查，如果窗口不满足一些条件，就不会再执行下面的代码逻辑。
- WindowToken 相关的处理，比如有的窗口类型需要提供 WindowToken，没有提供的话就不会执行下面的代码逻辑，有的窗口类型则需要由 WMS 隐式创建 WindowToken。
- WindowState 的创建和相关处理，将 WindowToken 和 WindowState 相关联。
- 创建和配置 DisplayContent，完成窗口添加到系统前的准备工作。

8.5 Window的删除过程

和7.4节中 Window 的创建和更新过程一样,要删除 Window 需要先调用 WindowManagerImpl 的 removeView 方法,在 removeView 方法中又会调用 WindowManagerGlobal 的 removeView 方法,我们就从这里开始讲起。为了表述得更易于理解,本节将要删除的 Window (View) 简称为 V。WindowManagerGlobal 的 removeView 方法如下所示:

frameworks/base/core/java/android/view/WindowManagerGlobal.java

```
public void removeView(View view, boolean immediate) {
    if (view == null) {
        throw new IllegalArgumentException("view must not be null");
    }
    synchronized (mLock) {
        int index = findViewLocked(view, true); //1
        View curView = mRoots.get(index).getView();
        removeViewLocked(index, immediate); //2
        if (curView == view) {
            return;
        }
        throw new IllegalStateException("Calling with view " + view
            + " but the ViewAncestor is attached to " + curView);
    }
}
```

在注释1处找到要 V 在 View 列表中的索引,在注释2处调用了 removeViewLocked 方法并将这个索引传进去,如下所示:

frameworks/base/core/java/android/view/WindowManagerGlobal.java

```
private void removeViewLocked(int index, boolean immediate) {
    ViewRootImpl root = mRoots.get(index); //1
    View view = root.getView();
    if (view != null) {
        InputMethodManager imm = InputMethodManager.getInstance(); //2
        if (imm != null) {
            imm.windowDismissed(mViews.get(index).getWindowToken()); //3
        }
    }
    boolean deferred = root.die(immediate); //4
    if (view != null) {
        view.assignParent(null);
        if (deferred) {
            mDyingViews.add(view);
        }
    }
}
```

```

    }
}
}

```

在注释 1 处根据传入的索引在 ViewRootImpl 列表中获得 V 的 ViewRootImpl。在注释 2 处得到 InputMethodManager 实例,如果 InputMethodManager 实例不为 null 则在注释 3 处调用 InputMethodManager 的 windowDismissed 方法来结束 V 的输入法相关的逻辑。在注释 4 处调用 ViewRootImpl 的 die 方法,如下所示:

frameworks/base/core/java/android/view/ViewRootImpl.java

```

boolean die(boolean immediate) {
    //die 方法需要立即执行并且此时 ViewRootImpl 不再执行 performTraversals 方法
    if (immediate && !mIsInTraversal) { //1
        doDie(); //2
        return false;
    }
    if (!mIsDrawing) {
        destroyHardwareRenderer();
    } else {
        Log.e(mTag, "Attempting to destroy the window while drawing!\n" +
            " window=" + this + ", title=" + mWindowAttributes.getTitle());
    }
    mHandler.sendMessage(MSG_DIE);
    return true;
}

```

在注释 1 处如果 immediate 为 true (需要立即执行), 并且 mIsInTraversal 值为 false 则执行注释 2 处的代码, mIsInTraversal 在执行 ViewRootImpl 的 performTraversals 方法时会被设置为 true, 在 performTraversals 方法执行完时被设置为 false, 因此注释 1 处可以理解为 die 方法需要立即执行并且此时 ViewRootImpl 不再执行 performTraversals 方法。在注释 2 处的 doDie 方法如下所示:

frameworks/base/core/java/android/view/ViewRootImpl.java

```

void doDie() {
    //检查执行 doDie 方法的线程的正确性
    checkThread(); //1
    if (LOCAL_LOGV) Log.v(mTag, "DIE in " + this + " of " + mSurface);
    synchronized (this) {
        if (mRemoved) { //2
            return;
        }
        mRemoved = true; //3
    }
}

```



```

if (mAdded) { //4
    dispatchDetachedFromWindow(); //5
}
if (mAdded && !mFirst) { //6
    destroyHardwareRenderer();
    if (mView != null) {
        int viewVisibility = mView.getVisibility();
        boolean viewVisibilityChanged = mViewVisibility != viewVisibility;
        if (mWindowAttributesChanged || viewVisibilityChanged) {
            try {
                if ((relayoutWindow(mWindowAttributes, viewVisibility, false)
                    & WindowManagerGlobal.RELAYOUT_RES_FIRST_TIME) != 0) {
                    mWindowSession.finishDrawing(mWindow);
                }
            } catch (RemoteException e) {
            }
        }
        mSurface.release();
    }
}
mAdded = false;
}
WindowManagerGlobal.getInstance().doRemoveView(this); //7
}

```

在注释 1 处用于检查执行 doDie 方法的线程的正确性，在注释 1 处的内部会判断执行 doDie 方法线程是否是创建 V 的原始线程，如果不是就会抛出异常，这是因为只有创建 V 的原始线程才能够操作 V。注释 2 到注释 3 处的代码用于防止 doDie 方法被重复调用。在注释 4 处 V 有子 View 就会调用注释 5 处的 dispatchDetachedFromWindow 方法来销毁 View。在注释 6 处如果 V 有子 View 并且不是第一次被添加，就会执行后面的代码逻辑。注释 7 处的 WindowManagerGlobal 的 doRemoveView 方法，如下所示：

frameworks/base/core/java/android/view/WindowManagerGlobal.java

```

void doRemoveView(ViewRootImpl root) {
    synchronized (mLock) {
        final int index = mRoots.indexOf(root); //1
        if (index >= 0) {
            mRoots.remove(index);
            mParams.remove(index);
            final View view = mViews.remove(index);
            mDyingViews.remove(view);
        }
    }
}

```

```

        if (ThreadedRenderers.sTrimForeground && ThreadedRenderers.isAvailable()) {
            doTrimForeground();
        }
    }
}

```

在 7.4.1 节我们知道了在 WindowManagerGlobal 中维护了和 Window 操作相关的三个列表，doRemoveView 方法会从这三个列表中清除 V 对应的元素。在注释 1 处找到 V 对应的 ViewRootImpl 在 ViewRootImpl 列表中的索引，接着根据这个索引从 ViewRootImpl 列表、布局参数列表和 View 列表中删除与 V 对应的元素。我们接着回到 ViewRootImpl 的 doDie 方法，查看注释 5 处的 dispatchDetachedFromWindow 方法做了什么：

frameworks/base/core/java/android/view/ViewRootImpl.java

```

void dispatchDetachedFromWindow() {
    ...
    try {
        mWindowSession.remove(mWindow);
    } catch (RemoteException e) {
    }
    ...
}

```

在 dispatchDetachedFromWindow 方法中主要调用了 IWindowSession 的 remove 方法，IWindowSession 在 Server 端的实现为 Session，不理解的读者请查看 7.4.1 节，这里不再赘述。Session 的 remove 方法如下所示：

frameworks/base/services/core/java/com/android/server/wm/Session.java

```

public void remove(IWindow window) {
    mService.removeWindow(this, window);
}

```

接着查看 WMS 的 removeWindow 方法：

frameworks/base/services/core/java/com/android/server/wm/WindowManagerService.java

```

void removeWindow(Session session, IWindow client) {
    synchronized(mWindowMap) {
        WindowState win = windowForClientLocked(session, client, false); //1
        if (win == null) {
            return;
        }
        win.removeIfPossible(); //2
    }
}

```

在注释 1 处用于获取 Window 对应的 WindowState, WindowState 用于保存窗口的信息, 在 WMS 中它用来描述一个窗口。接着在注释 2 处调用 WindowState 的 removeIfPossible 方法, 如下所示:

```
frameworks/base/services/core/java/com/android/server/wm/WindowState.java
```

```
@Override
void removeIfPossible() {
    super.removeIfPossible();
    removeIfPossible(false /*keepVisibleDeadWindow*/);
}
```

又会调用 removeIfPossible 方法, 如下所示:

```
frameworks/base/services/core/java/com/android/server/wm/WindowState.java
```

```
private void removeIfPossible(boolean keepVisibleDeadWindow) {
    ...条件判断过滤, 满足其中一个条件就会 return, 推迟删除操作
    removeImmediately();//1
    if (wasVisible && mService.updateOrientationFromAppTokensLocked(false,
        displayId)) {
        mService.mH.obtainMessage(SEND_NEW_CONFIGURATION, displayId).
            sendToTarget();
    }
    mService.updateFocusedWindowLocked(UPDATE_FOCUS_NORMAL, true /*update
        InputWindows*/);
    Binder.restoreCallingIdentity(origId);
}
```

removeIfPossible 方法和它的名字一样, 并不是直接执行删除操作的, 而是进行多个条件判断过滤, 满足其中一个条件就会 return, 推迟删除操作。比如 V 正在运行一个动画, 这时就得推迟删除操作, 直到动画完成。通过这些条件判断过滤就会执行注释 1 处的 removeImmediately 方法:

```
frameworks/base/services/core/java/com/android/server/wm/WindowState.java
```

```
@Override
void removeImmediately() {
    super.removeImmediately();
    if (mRemoved) {//1
        if (DEBUG_ADD_REMOVE) Slog.v(TAG_WM,
            "WS.removeImmediately: " + this + " Already removed...");
        return;
    }
    mRemoved = true;//2
    ...
}
```

```

        mPolicy.removeWindowLw(this); //3
        disposeInputChannel();
        mWinAnimator.destroyDeferredSurfaceLocked();
        mWinAnimator.destroySurfaceLocked();
        mSession.windowRemovedLocked(); //4
        try {
            mClient.asBinder().unlinkToDeath(mDeathRecipient, 0);
        } catch (RuntimeException e) {
        }
        mService.postWindowRemoveCleanupLocked(this); //5
    }

```

`removeImmediately` 方法如同它的名字一样，用于立即进行删除操作。在注释 1 处的 `mRemoved` 为 `true` 意味着正在执行删除 Window 操作，在注释 1 到注释 2 处之间的代码用于防止重复删除操作。在注释 3 处如果当前要删除的 Window 是 `StatusBar` 或者 `NavigationBar` 就会将这个 Window 从对应的控制器中删除。在注释 4 处将 `V` 对应的 `Session` 从 WMS 的 `ArraySet<Session> mSessions` 中删除并清除 `Session` 对应的 `SurfaceSession` 资源（`SurfaceSession` 是 `SurfaceFlinger` 的一个连接，通过这个连接可以创建 1 个或者多个 `Surface` 并渲染到屏幕上）。在注释 5 处调用了 WMS 的 `postWindowRemoveCleanupLocked` 方法用于对 `V` 进行一些集中的清理工作，这里就不再继续深挖下去，有兴趣的读者可以自行查看源码。

Window 的删除过程就讲到这里，虽然删除的操作逻辑比较复杂，但是可以简单地总结为以下 4 点：

- 检查删除线程的正确性，如果不正确就抛出异常。
- 从 `ViewRootImpl` 列表、布局参数列表和 `View` 列表中删除与 `V` 对应的元素。
- 判断是否可以直接执行删除操作，如果不能就推迟删除操作。
- 执行删除操作，清理和释放与 `V` 相关的一切资源。

8.6 本章小结

在这一章我们学习了 WMS 的职责以及 Window 的添加和删除过程等知识点，从 WMS 的职责可以看出 WMS 很复杂，与它关联的有窗口管理、窗口动画、输入系统和 `Surface`，它们每一个都是重要且复杂的系统，本章只介绍其中的窗口管理，因为它和应用开发的关联比较紧密，至于其他 WMS 关联的系统则需要读者去阅读其他一些专业的剖析系统源码的书籍和资料了。

第 9 章

JNI 原理

JNI 是 Java Native Interface 的缩写，译为 Java 本地接口，是 Java 与其他语言通信的桥梁。当出现一些用 Java 无法处理的任务时，开发人员就可以使用 JNI 技术来完成。一般来说主要有以下情况需要用到 JNI 技术。

- 需要调用 Java 语言不支持的依赖于操作系统平台特性的一些功能。例如：需要调用当前的 UNIX 系统的某个功能，而 Java 不支持这个功能，就需要用到 JNI 技术来实现。
- 为了整合一些以前的非 Java 语言开发的系统。例如：需要用到早期实现的 C/C++ 语言开发的一些功能或系统，将这些功能整合到当前的系统或新的版本中。
- 为了节省程序的运行时间，必须采用其他语言（比如 C/C++ 语言）来提升运行效率。例如：游戏、音视频开发涉及的音视频编解码和图像绘制需要更快的处理速度。

JNI 不只是应用于 Android 开发，它有着非常广泛的应用场景。JNI 在 Android 中的应用场景也十分广泛，主要有音视频开发、热修复和插件化、逆向开发、系统源码调用等。为了方便地使用 JNI 技术，Android 还提供了 NDK 这个工具集合，NDK 开发是基于 JNI 的，它和 JNI 开发本质上并没有区别，理解了 JNI 原理，NDK 开发也会很容易掌握。本章不会介绍 NDK 开发的知识，而是从系统源码调用的角度对 JNI 原理进行分析。本章介绍 JNI 主要有两大用意：一个是便于理解系统源码，如果不理解 JNI 原理的话，那么技术视野就会很容易停留在 Java Framework 层；另一个就是为讲解热修复和插件化的原理做好知识铺垫。从这两大用意可以看出在普通的 Android 应用开发中几乎不会涉及 JNI，因此本章不

会过于深入地讲解 JNI，如果想要更加深入的学习可以阅读专门介绍 JNI 和 NDK 的专业书籍和文章。

9.1 系统源码中的JNI

Android 系统按语言来划分的话由两个世界组成，分别是 Java 世界和 Native 世界。为什么要这么划分呢？Android 系统由 Java 编写不好吗？除了性能的原因外，最主要的原因就是在 Java 诞生之前，就有很多程序和库都是由 Native 语言写的，因此，重复利用这些 Native 语言编写的库是十分必要的，况且 Native 语言编写的库具有更好的性能。这样就产生了一个问题，Java 世界的代码要怎么使用 Native 世界的代码呢？这就需要有一个桥梁来将它们连接在一起，而 JNI 就是这个桥梁，如图 9-1 所示。

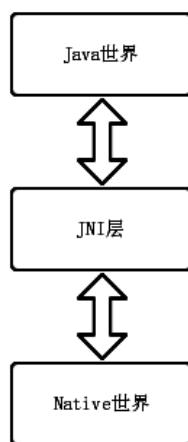


图 9-1 JNI 桥梁

通过 JNI，Java 世界的代码就可以访问 Native 世界的代码，同样地，Native 世界的代码也可以访问 Java 世界的代码。为了讲解 JNI 我们需要分析系统的源码，在前作《Android 进阶之光》一书的最后一章中我拿 MediaPlayer 框架做了举例，这里换 MediaRecorder 框架来举例，它和 MediaPlayer 框架的调用过程十分类似。

9.2 MediaPlayer框架中的JNI

MediaPlayer 我们应该都不陌生，它用于录音和录像。这里不会主要介绍 MediaPlayer 框架，而重点介绍的是 MediaPlayer 框架中的 JNI，如图 9-2 所示。

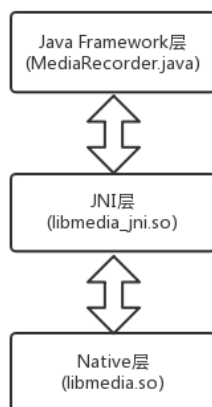


图 9-2 MediaPlayer 框架中的 JNI

如图 9-1 所示，Java Framework 层对应的是 MediaPlayer.java，也就是我们在应用开发中直接调用的类。JNI 层对应的是 libmedia_jni.so，可以看到这个动态库的名称含有“_jni”，这说明它是一个 JNI 的动态库。Native 层对应的是 libmedia.so，这个动态库完成了实际的调用的功能。接下来我们分别学习 Java Framework 层和 JNI 层的 MediaPlayer。

9.2.1 Java Framework层的MediaPlayer

我们先来查看 MediaPlayer.java 的源码，截取部分和 JNI 有关的源码如下所示：

```

frameworks/base/media/java/android/media/MediaRecorder.java

public class MediaRecorder{
    static {
        System.loadLibrary("media_jni");//1
        native_init();//2
    }
    ...
    private static native final void native_init();//3
    ...
    public native void start() throws IllegalStateException;
    ...
}
  
```

在静态代码块中首先调用了注释 1 处的代码，用来加载名为 `media_jni` 的动态库，也就是 `libmedia_jni.so`。接着调用注释 2 处的 `native_init` 方法，其内部会调用 Native 方法，用来完成 JNI 的注册。注释 3 处的 `native_init` 方法用 `native` 来修饰，说明它是一个 native 方法，表示由 JNI 来实现。`MediaRecorder` 的 `start` 方法同样也是一个 native 方法。对于 Java Framework 层来说只需要加载对应的 JNI 库，接着声明 native 方法就可以了，剩下的工作由 JNI 层来完成。

9.2.2 JNI层的MediaRecorder

`MediaRecorder` 的 JNI 层由 `android_media_recorder.cpp` 实现，native 方法 `native_init` 和 `start` 的 JNI 层实现如下所示：

```
frameworks/base/media/jni/android_media_MediaRecorder.cpp

static void
android_media_MediaRecorder_native_init(JNIEnv *env)
{
    jclass clazz;
    clazz = env->FindClass("android/media/MediaRecorder");
    if (clazz == NULL) {
        return;
    }
    ...
    fields.post_event = env->GetStaticMethodID(clazz, "postEventFromNative",
                                                "(Ljava/lang/Object;IIILjava/lang/Object;)V");

    if (fields.post_event == NULL) {
        return;
    }
}

static void
android_media_MediaRecorder_start(JNIEnv *env, jobject thiz)
{
    ALOGV("start");
    sp<MediaRecorder> mr = getMediaRecorder(env, thiz);
    process_media_recorder_call(env, mr->start(), "java/lang/RuntimeException",
                                "start failed.");
}
```

`android_media_MediaRecorder_native_init` 方法是 `native_init` 方法在 JNI 层的实现，`android_media_MediaRecorder_start` 方法则是 `start` 方法在 JNI 层的实现。那么 `native_init` 方法是如何找到对应的 `android_media_MediaRecorder_native_init` 方法的呢？这就需要了解 JNI 方法注册的知识。

9.2.3 Native方法注册

Native 方法注册分为静态注册和动态注册，其中静态注册多用于 NDK 开发，而动态注册多用于 Framework 开发，下面分别对这两种注册方式进行讲解。

9.2.3.1 静态注册

在 Android Studio (AS) 中新建一个 Java Library，命名为 media，这里仿照系统的 MediaRecorder.java，写一个简单的 MediaRecorder.java，如下所示：

```
media/src/main/java/MediaRecorder.java
```

```
package com.example;
public class MediaRecorder {
    static {
        System.loadLibrary("media_jni");
        native_init();
    }
    private static native final void native_init();
    public native void start() throws IllegalStateException;
}
```

这里不要在乎上段代码在 AS 中会标红：“找不到 native_init 和 start 方法所对应 JNI 方法”，我们的主要目的是来查看 Native 方法是如何注册的，AS 标红的问题并不是主要的。现在我们对 MediaRecorder.java 进行编译和生成 JNI 方法，进入项目的 media/src/main/java 目录中执行如下命令：

```
javac com/example/MediaRecorder.java
javah com.example.MediaRecorder
```

第二个命令会在当前目录 (media/src/main/java) 中生成 com_example_MediaRecorder.h 文件，如下所示：

```
media/src/main/java/com_example_MediaRecorder.h
```

```
/* DO NOT EDIT THIS FILE - it is machine generated */
#include <jni.h>
/* Header for class com_example_MediaRecorder */
#ifndef _Included_com_example_MediaRecorder
#define _Included_com_example_MediaRecorder
#ifdef __cplusplus
extern "C" {
#endif
/*
```

```

* Class:    com_example_MediaRecorder
* Method:   native_init
* Signature: ()V
*/
JNIEXPORT void JNICALL Java_com_example_MediaRecorder_native_init
    (JNIEnv *, jclass); //1
/*
* Class:    com_example_MediaRecorder
* Method:   start
* Signature: ()V
*/
JNIEXPORT void JNICALL Java_com_example_MediaRecorder_start
    (JNIEnv *, jobject);
#ifdef __cplusplus
}
#endif
#endif

```

nativeinit 方法被声明为注释 1 处的方法，也就是 Java_com_example_MediaRecorder_native_init，以“Java”开头说明是在 Java 平台中调用 JNI 方法的，后面的 com_example_MediaRecorder_native_init 指的是包名+类名+方法名的格式，原本在 Java 中应该是以“.”来进行分割，这里却用了“ ”（空格），这是因为在 Native 语言中“.”有特殊的含义。还有眼尖的读者发现了注释 1 处的方法名还多了一个“1”，这是因为 Java 的 native_init 方法中包含了“_”，转换成 JNI 方法后会变成“_1”。其中 JNIEnv 是 Native 世界中 Java 环境的代表，通过 JNIEnv * 指针就可以在 Native 世界中访问 Java 世界的代码进行操作，它只在创建它的线程中有效，不能跨线程传递。jclass 是 JNI 的数据类型，对应 Java 的 java.lang.Class 实例。jobject 同样也是 JNI 的数据类型，对应于 Java 的 Object，关于 JNIEnv 和 JNI 的数据类型会在后面进行介绍。

当我们在 Java 中调用 native_init 方法时，就会从 JNI 中寻找 Java_com_example_MediaRecorder_native_init 函数，如果没有就会报错，如果找到就会为 native_init 和 Java_com_example_MediaRecorder_native_init 建立关联，其实是保存 JNI 的函数指针，这样再次调用 native_init 方法时直接使用这个函数指针就可以了。静态注册就是根据方法名，将 Java 方法和 JNI 函数建立关联，但是它有一些缺点：

- JNI 层的函数名称过长。
- 声明 Native 方法的类需要用 javah 生成头文件。
- 初次调用 Native 方法时需要建立关联，影响效率。

我们知道,静态注册就是 Java 的 Native 方法通过方法指针来与 JNI 进行关联,如果 Java 的 Native 方法知道它在 JNI 中对应的函数指针,就可以避免上述的缺点,这就是动态注册。

9.2.3.2 动态注册

JNI 中有一种结构用来记录 Java 的 Native 方法和 JNI 方法的关联关系,它就是 `JNINativeMethod`,它在 `jni.h` 中被定义:

```
typedef struct {
    const char* name;//Java 方法的名字
    const char* signature;//Java 方法的签名信息
    void*      fnPtr;//JNI 中对应的方法指针
} JNINativeMethod;
```

系统的 `MediaRecorder` 采用的就是动态注册,我们来查看它的 JNI 层是怎么做的:

`frameworks/base/media/jni/android_media_MediaRecorder.cpp`

```
static const JNINativeMethod gMethods[] = {
...
    {"start",          "()V",    (void *)android_media_MediaRecorder_start},//1
    {"stop",           "()V",    (void *)android_media_MediaRecorder_stop},
    {"pause",          "()V",    (void *)android_media_MediaRecorder_pause},
    {"resume",         "()V",    (void *)android_media_MediaRecorder_resume},
    {"native_reset",   "()V",    (void *)android_media_MediaRecorder_native_reset},
    {"release",        "()V",    (void *)android_media_MediaRecorder_release},
    {"native_init",    "()V",    (void *)android_media_MediaRecorder_native_init},
...
};
```

上面定义了一个 `JNINativeMethod` 类型的 `gMethods` 数组,里面存储的就是 `MediaRecorder` 的 Native 方法与 JNI 层函数的对应关系,其中注释 1 处“start”是 Java 层的 Native 方法,它对应的 JNI 层的函数为 `android_media_MediaRecorder_start`。“()V”是 start 方法的签名信息,关于 Java 方法的签名信息后面会介绍。只定义 `JNINativeMethod` 类型的数组是没有用的,还需要注册它,注册的函数为 `register_android_media_MediaRecorder`,这个函数会在哪里调用呢?答案是在 `register_android_media_MediaRecorder` 函数的英文注释上: `JNI_OnLoad` in `android_media_MediaPlayer.cpp`。这个 `JNI_OnLoad` 函数会在调用 `System.loadLibrary` 函数后调用,因为多媒体框架中的很多框架都要进行 `JNINativeMethod` 数组注册,因此,注册函数就被统一定义在 `android_media_MediaPlayer.cpp` 的 `JNI_OnLoad` 函数中,如下所示:

```
frameworks/base/media/jni/android_media_MediaPlayer.cpp
```

```
jint JNI_OnLoad(JavaVM* vm, void* /* reserved */)
{
    JNIEnv* env = NULL;
    jint result = -1;
    if (vm->GetEnv((void**) &env, JNI_VERSION_1_4) != JNI_OK) {
        ALOGE("ERROR: GetEnv failed\n");
        goto *bail;
    }
    assert(env != NULL);
    ...
    if (register_android_media_MediaPlayer(env) < 0) { //1
        ALOGE("ERROR: MediaPlayer native registration failed\n");
        goto *bail;
    }
    if (register_android_media_MediaRecorder(env) < 0) { //2
        ALOGE("ERROR: MediaRecorder native registration failed\n");
        goto *bail;
    }
    ...
    result = JNI_VERSION_1_4;
bail:
    return result;
}
```

在 `JNI_OnLoad` 函数中调用了整个多媒体框架的注册 `JNINativeMethod` 数组的函数，在注释 2 处调用了 `register_android_media_MediaRecorder` 函数，同样地，在注释 1 处 `MediaPlayer` 框架的注册 `JNINativeMethod` 数组的函数 `register_android_media_MediaPlayer` 也被调用了。`register_android_media_MediaRecorder` 函数如下所示：

```
frameworks/base/media/jni/android_media_MediaRecorder.cpp
```

```
//JNI_OnLoad in android_media_MediaPlayer.cpp
int register_android_media_MediaRecorder(JNIEnv *env)
{
    return AndroidRuntime::registerNativeMethods(env,
        "android/media/MediaRecorder", gMethods, NELEM(gMethods));
}
```

在 `register_android_media_MediaRecorder` 方法中返回了 `AndroidRuntime` 的 `registerNativeMethods` 函数，如下所示：

```
frameworks/base/core/jni/AndroidRuntime.cpp
```

```
/*static*/ int AndroidRuntime::registerNativeMethods(JNIEnv* env,
    const char* className, const JNINativeMethod* gMethods, int numMethods)
{
    return jniRegisterNativeMethods(env, className, gMethods, numMethods);
}
```

在 registerNativeMethods 函数中又返回了 jniRegisterNativeMethods 函数，它被定义在 JNI 帮助类 JNIHelp.cpp 中：

```
libnativehelper/JNIHelp.cpp
```

```
extern "C" int jniRegisterNativeMethods(C_JNIEnv* env, const char* className,
    const JNINativeMethod* gMethods, int numMethods)
{
    ...
    if ((*env)->RegisterNatives(e, c.get(), gMethods, numMethods) < 0) { //1
        char* tmp;
        const char* msg;
        if (asprintf(&tmp, "RegisterNatives failed for '%s'; aborting...", className)
            == -1) {
            msg = "RegisterNatives failed; aborting...";
        } else {
            msg = tmp;
        }
        e->FatalError(msg);
    }
    return 0;
}
```

从注释 1 处可以看出，最终通过调用的 JNIEnv 的 RegisterNatives 函数来完成 JNI 的注册，JNIEnv 在 JNI 中十分重要，后面会介绍它。动态注册就讲到这里，可以看出动态注册要比静态注册复杂一些，但是一劳永逸。

9.3 数据类型的转换

JNI 数据类型的转换的例子仍旧以上文讲到的 MediaRecorder 来举例，先来查看 android_media_MediaRecorder.cpp 中的 android_media_MediaRecorder_start 函数，如下所示：

frameworks/base/media/jni/android_media_MediaRecorder.cpp

```
static void
android_media_MediaRecorder_start(JNIEnv *env, jobject this)
{
    ALOGV("start");
    sp<MediaRecorder> mr = getMediaRecorder(env, this);
    process_media_recorder_call(env, mr->start(), "java/lang/RuntimeException",
    "start failed.");
}
```

android_media_MediaRecorder_start 函数的第二个参数为 jobject 类型，它是 JNI 层的数据类型，Java 的数据类型到了 JNI 层就需要转换为 JNI 层的数据类型。Java 的数据类型分为基本数据类型和引用数据类型，JNI 层对于这两种类型也做了区分，我们先来查看基本数据类型的转换。

9.3.1 基本数据类型的转换

从表 9-1 可以看出，基本数据类型转换，除了最后一行的 void，其他的数据类型只需要在前面加上“j”就可以了。第三列的 Signature 代表签名格式，后面会介绍它，接着来看引用数据类型的转换。

表 9-1 基本数据类型的转换

Java	Native	Signature
byte	jbyte	B
char	jchar	C
double	jdouble	D
float	jfloat	F
int	jint	I
short	jshort	S
long	jlong	J
boolean	jboolean	Z
void	void	V

9.3.2 引用数据类型的转换

从表 9-2 可以看出，数组的 JNI 层数据类型需要以“Array”结尾，签名格式的开头都会有“[”。需要注意有些数据类型的签名以“;”结尾，引用数据类型还具有继承关系，如图 9-3 所示。

表 9-2 引用数据类型的转换

Java	Native	Signature
所有对象	jobject	L+classname +;
Class	jclass	Ljava/lang/Class;
String	jstring	Ljava/lang/String;
Throwable	jthrowable	Ljava/lang/Throwable;
Object[]	jobjectArray	[L+classname +;
byte[]	jbyteArray	[B
char[]	jcharArray	[C
double[]	jdoubleArray	[D
float[]	jfloatArray	[F
int[]	jintArray	[I
short[]	jshortArray	[S
long[]	jlongArray	[J
boolean[]	jbooleanArray	[Z

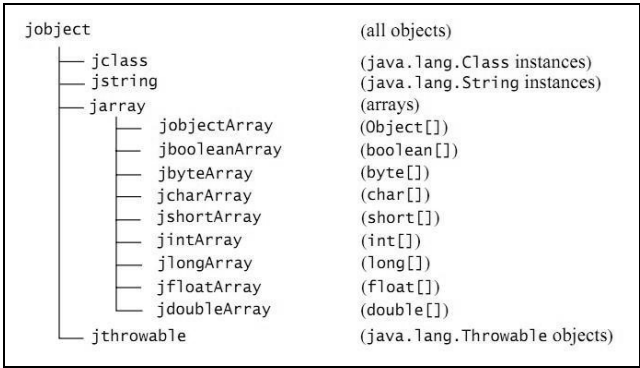


图 9-3 引用数据类型的继承关系

从图 9-3 可以看出 jclass、jstring、jarray 和 jthrowable 都继承 jobject，而 jobjectArray、jintArray 和 jlongArray 等类型都继承 jarray。

关于引用数据类型，这里接着列举 MediaRecorder 框架的 Java 方法，如下所示：

frameworks/base/media/java/android/media/MediaRecorder.java

```
private native void _setOutputFile(FileDescriptor fd, long offset, long length)
    throws IllegalStateException, IOException
```

_setOutputFile 方法对应的 JNI 层的方法为：

```
frameworks/base/media/jni/android_media_MediaRecorder.cpp
```

```
static void
android_media_MediaRecorder_setOutputFileFD(JNIEnv *env, jobject thiz, jobject
fileDescriptor, jlong offset, jlong length)
{
    ...
}
```

对比这两个方法可以看到，FileDescriptor 类型转换为了 jobject 类型，long 类型转换为了 jlong 类型。

9.4 方法签名

在表 9-1 和表 9-2 中列举了数据类型的签名格式 (Signature)，方法签名就是由签名格式组成的，那么方法签名有什么作用呢？我们看下面的代码：

```
frameworks/base/media/jni/android_media_MediaRecorder.cpp
```

```
static const JNINativeMethod gMethods[] = {
    ...
    {"native_init", "()V", (void *)android_media_MediaRecorder_native_init},
    {"native_setup", "(Ljava/lang/Object;Ljava/lang/String;Ljava/lang/String;)V",
    (void *)android_media_MediaRecorder_native_setup},
    ...
};
```

gMethods 数组中存储的是 MediaRecorder 的 Native 方法与 JNI 层函数的对应关系，其中 “()V” 和 “(Ljava/lang/Object;Ljava/lang/String;Ljava/lang/String;)V” 就是方法签名。Java 是有重载方法的，可以定义方法名相同，但参数不同的方法，正因为如此，在 JNI 中仅仅通过方法名是无法找到 Java 中对应的具体方法的，JNI 为了解决这一问题就将参数类型和返回值类型组合在一起作为方法签名。通过方法签名和方法名就可以找到对应的 Java 方法。JNI 的方法签名的格式为：

(参数签名格式...) 返回值签名格式

以上面 gMethods 数组的 native_setup 函数举例，它在 Java 中是如下定义的：

```
private native final void native_setup(Object mediarecorder_this,
    String clientName, String opPackageName) throws IllegalStateException;
```


它在 JNI 中的方法签名为：

```
(Ljava/lang/Object;Ljava/lang/String;Ljava/lang/String;)V
```

参照表 9-2, native_setup 函数的第一个参数的签名为 “Ljava/lang/Object;”, 后两个参数的签名为 “Ljava/lang/String;”, 返回值类型 void 的签名为 “V”, 组合起来就是上面的方法签名。如果我们每次编写 JNI 时都要组织方法签名, 也会是一件比较头疼的事, 而且也容易出错, 幸好 Java 提供了 javap 命令来自动生成方法签名。我们先写一个简单的 MediaRecorder.java 包含上面的 native_setup 方法:

```
public class MediaRecorder {
    static {
        System.loadLibrary("media_jni");
        native_init();
    }
    private static native final void native_init();
    private native final void native_setup(Object mediarecorder_this,
        String clientName, String opPackageName) throws IllegalStateException;
}
```

这个 Java 文件的本地地址为 D:/Android/MediaRecorder.java, 接着执行如下命令:

```
javac D:/Android/MediaRecorder.java
```

执行命令后会生成 MediaRecorder.class 文件, 最后使用 javap 命令:

```
javap -s -p D:/Android/MediaRecorder.class
```

其中 s 表示输出内部类型签名, p 表示打印出所有的方法和成员 (默认打印 public 成员), 最终在 cmd 中的打印结果如图 9-4 所示。

```
C:\Users\Administrator.EIT-20160520RHS>javap -s -p D:\Android\MediaRecorder.class
s
Compiled from "MediaRecorder.java"
public class MediaRecorder {
    public MediaRecorder();
        descriptor: <()U

    private static final native void native_init();
        descriptor: <()U

    private final native void native_setup<Ljava/lang/Object, Ljava/lang/String, java
a.lang.String> throws java.lang.IllegalStateException;
        descriptor: <(Ljava/lang/Object;Ljava/lang/String;Ljava/lang/String;)U

    static {};
        descriptor: <()U
}
```

图 9-4 打印结果

从图 9-4 可以很清晰地看到输出的 native_setup 方法的签名和此前给出的一致。

9.5 解析JNIEnv

JNIEnv 是 Native 世界中 Java 环境的代表，通过 JNIEnv *指针就可以在 Native 世界中访问 Java 世界的代码进行操作，它只在创建它的线程中有效，不能跨线程传递，因此不同线程的 JNIEnv 是彼此独立的，JNIEnv 的主要作用有以下两点：

- 调用 Java 的方法。
- 操作 Java（操作 Java 中的变量和对象等）。

先来看 JNIEnv 的定义，如下所示：

```
libnativehelper/include/nativehelper/jni.h

#ifdef __cplusplus
//C++中 JNIEnv 的类型
typedef _JNIEnv JNIEnv; //1
typedef _JavaVM JavaVM;
#else
//C 中 JNIEnv 的类型
typedef const struct JNINativeInterface* JNIEnv; //2
typedef const struct JNIInvokeInterface* JavaVM;
#endif
```

这里使用预定义宏__cplusplus 来区分 C 和 C++两种代码，如果定义了__cplusplus，就是 C++代码中的定义，否则就是 C 代码中的定义。在这里我们也看到了 JavaVM，它是虚拟机在 JNI 层的代表，在一个虚拟机进程中只有一个 JavaVM，因此，该进程的所有线程都可以使用这个 JavaVM。通过 JavaVM 的 AttachCurrentThread 函数可以获取这个线程的 JNIEnv，这样就可以在不同的线程中调用 Java 方法了。还要记得在使用 AttachCurrentThread 函数的线程退出前，务必要调用 DetachCurrentThread 函数来释放资源。

从注释 2 处可以看出在 C 中，JNIEnv 类型是 JNINativeInterface*，从注释 1 处可以看出在 C++中 JNIEnv 的类型是 JNIEnv，JNIEnv 是如何定义的呢？如下所示：

```
libnativehelper/include/nativehelper/jni.h

struct _JNIEnv {
    /* do not rename this; it does not seem to be entirely opaque */
    const struct JNINativeInterface* functions;
#ifdef __cplusplus
    ...
    jclass FindClass(const char* name)
    { return functions->FindClass(this, name); }
#endif
};
```

```

...
jmethodID GetMethodID(jclass clazz, const char* name, const char* sig)
{ return functions->GetMethodID(this, clazz, name, sig); }
...
jfieldID GetFieldID(jclass clazz, const char* name, const char* sig)
{ return functions->GetFieldID(this, clazz, name, sig); }
...
}

```

`_JNIEnv` 是一个结构体，其内部又包含了 `JNINativeInterface`。在 `_JNIEnv` 中定义了很多函数，这里列举了 3 个比较常用的函数，`FindClass` 用来找到 Java 中指定名称的类，`GetMethodID` 用来得到 Java 中的方法，`GetFieldID` 用来得到 Java 中的成员变量，这里可以发现这三个函数都调用了 `JNINativeInterface` 中定义的函数，因此可以得出结论，无论是 C 还是 C++，`JNIEnv` 的类型都和 `JNINativeInterface` 结构有关，`JNINativeInterface` 的定义如下所示：

```
libnativehelper/include/nativehelper/jni.h
```

```

struct JNINativeInterface {
...
jclass      (*FindClass)(JNIEnv*, const char*);
...
jmethodID   (*GetMethodID)(JNIEnv*, jclass, const char*, const char*);
...
jfieldID    (*GetFieldID)(JNIEnv*, jclass, const char*, const char*);
...
}

```

在 `JNINativeInterface` 结构中定义了很多和 `JNIEnv` 结构体对应的函数指针，这里只给出了上面 `JNIEnv` 结构体中对应的三个函数指针定义。通过这些函数指针的定义，就能够定位到虚拟机中的 JNI 函数表，从而实现了 JNI 层在虚拟机中的函数调用，这样 JNI 层就可以调用 Java 世界的方法了。

9.5.1 jfieldID和jmethodID

在 `_JNIEnv` 结构体中定义了很多函数，这些函数都会有不同的返回值，如下所示：

```
libnativehelper/include/nativehelper/jni.h
```

```

struct _JNIEnv {
    const struct JNINativeInterface* functions;
#ifdef __cplusplus

```

```

...
jmethodID GetMethodID(jclass clazz, const char* name, const char* sig)//1
{ return functions->GetMethodID(this, clazz, name, sig); }
...
jfieldID GetFieldID(jclass clazz, const char* name, const char* sig)
{ return functions->GetFieldID(this, clazz, name, sig); }
...
}

```

这里我们列举了两个函数，这两个函数的返回值分别为 `jmethodID` 和 `jfieldID`，当然还有其他很多返回值，比如 `jobject`、`jbooleanArray` 和 `jboolean` 等，这里我们拿 `jmethodID` 和 `jfieldID` 来进行举例，来查看这些返回值在 JNI 层的应用。`jfieldID` 和 `jmethodID` 分别用来代表 Java 类中的成员变量和方法。在注释 1 处 `jclass` 代表 Java 类，`name` 代表成员方法或者成员变量的名字，`sig` 为这个方法和变量的签名。我们来查看 `MediaRecorder` 框架的 JNI 层是如何使用 `GetMethodID` 和 `GetFieldID` 这两个方法的，如下所示：

```

frameworks/base/media/jni/android_media_MediaRecorder.cpp

static void
android_media_MediaRecorder_native_init(JNIEnv *env)
{
    jclass clazz;
    clazz = env->FindClass("android/media/MediaRecorder");//1
    if (clazz == NULL) {
        return;
    }
    fields.context = env->GetFieldID(clazz, "mNativeContext", "J");//2
    if (fields.context == NULL) {
        return;
    }
    fields.surface = env->GetFieldID(clazz, "mSurface", "Landroid/view/Surface;");//3
    if (fields.surface == NULL) {
        return;
    }
    jclass surface = env->FindClass("android/view/Surface");
    if (surface == NULL) {
        return;
    }
    fields.post_event = env->GetStaticMethodID(clazz, "postEventFromNative",
                                                "(Ljava/lang/Object;IIILjava/lang/Object;) V");//4
    if (fields.post_event == NULL) {

```

```

        return;
    }
}

```

在注释 1 处，通过 FindClass 来找到 Java 层的 MediaRecorder 的 Class 对象，并赋值给 jclass 类型的变量 clazz，因此，clazz 就是 Java 层的 MediaRecorder 在 JNI 层的代表。在注释 2 和注释 3 处的代码用来找到 Java 层的 MediaRecorder 中名为 mNativeContext 和 mSurface 的成员变量，并分别赋值给 context 和 surface。在注释 4 处获取 Java 层的 MediaRecorder 中名为 postEventFromNative 的静态方法，并赋值给 fields.post_event，其中 fields 的定义为：

```
frameworks/base/media/jni/android_media_MediaRecorder.cpp
```

```

struct fields_t {
    jfieldID  context;
    jfieldID  surface;
    jmethodID post_event;
};
static fields_t fields;

```

将这些成员变量和方法赋值给 jfieldID 和 jmethodID 类型的变量有两个原因：第一是为了效率考虑，如果每次调用相关方法时都要查询方法和变量，显然会效率很低；第二是这些成员变量和方法都是本地引用，在 android_media_MediaRecorder_native_init 函数返回时这些本地引用会被自动释放，因此用 fields 来进行保存，以便后续使用，关于本地引用会在 9.6 节中讲到。结合上面两方面原因，在 MediaRecorder 框架 JNI 层的初始化方法 android_media_MediaRecorder_native_init 中将这 jfieldID 和 jmethodID 类型的变量保存起来，是为了更高效地供后续使用。

9.5.2 使用jfieldID和jmethodID

保存了 jfieldID 和 jmethodID 类型的变量，接着怎么使用它们呢？如下所示：

```
frameworks/base/media/jni/android_media_MediaRecorder.cpp
```

```

void JNIRecorderListener::notify(int msg, int ext1, int ext2)
{
    ALOGV("JNIRecorderListener::notify");
    JNIEnv *env = AndroidRuntime::getJNIEnv();
    env->CallStaticVoidMethod(mClass, fields.post_event, mObject, msg, ext1, ext2,
        NULL); //1
}

```

在注释 1 处调用了 JNIEnv 的 CallStaticVoidMethod 函数，其中就传入了 fields.post_event，

从 9.5.1 节中我们得知，它其实是保存了 Java 层 `MediaRecorder` 的静态方法 `postEventFromNative`：

```
frameworks/base/media/java/android/media/MediaRecorder.java

private static void postEventFromNative(Object mediarecorder_ref,
                                       int what, int arg1, int arg2, Object obj)
{
    MediaRecorder mr = (MediaRecorder)((WeakReference)mediarecorder_ref).get();
    if (mr == null) {
        return;
    }
    if (mr.mEventHandler != null) {
        Message m = mr.mEventHandler.obtainMessage(what, arg1, arg2, obj); //1
        mr.mEventHandler.sendMessage(m); //2
    }
}
```

在注释 1 处会创建一个消息，在注释 2 处将这个消息发送给 `MediaRecorder` 内部类 `mEventHandler` 来处理，这样做的目的是将代码逻辑运行在应用程序的主线程中。`JNIEnv` 的 `CallStaticVoidMethod` 函数会调用 Java 层 `MediaRecorder` 的静态方法 `postEventFromNative`，也就是说 `JNIEnv` 的 `CallStaticVoidMethod` 函数可以访问 Java 的静态方法，同理如果想要访问 Java 的方法则可以使用 `JNIEnv` 的 `CallVoidMethod` 函数。上面的例子是使用了 `jmethodID`，接着来查看 `jfieldID` 是如何应用的：

```
frameworks/base/media/jni/android_media_MediaRecorder.cpp

static void
android_media_MediaRecorder_prepare(JNIEnv *env, jobject thiz)
{
    ALOGV("prepare");
    sp<MediaRecorder> mr = getMediaRecorder(env, thiz);
    jobject surface = env->GetObjectField(thiz, fields.surface); //1
    if (surface != NULL) {
        const sp<Surface> native_surface = get_surface(env, surface);
        ...
    }
    process_media_recorder_call(env, mr->prepare(), "java/io/IOException", "prepare
failed.");
}
```

在注释 1 处调用了 `JNIEnv` 的 `GetObjectField` 函数，参数中的 `fields.surface` 是 `jfieldID` 类型的变量，用来保存 Java 层 `MediaRecorder` 中的成员变量 `mSurface`，`mSurface` 的类型为 `Surface`，这样通过 `GetObjectField` 函数就得到了 `mSurface` 在 JNI 层中对应的 `jobject` 类型变量 `surface`。

9.6 引用类型

和 Java 的引用类型一样，JNI 也有引用类型，它们分别是本地引用（Local References）、全局引用（Global References）和弱全局引用（Weak Global References），下面分别介绍它们。

9.6.1 本地引用

JNIEnv 提供的函数所返回的引用基本上都是本地引用，因此本地引用也是 JNI 中最常见的引用类型。本地引用的特点主要有以下几点：

- 当 Native 函数返回时，这个本地引用就会被自动释放。
- 只在创建它的线程中有效，不能够跨线程使用。
- 局部引用是 JVM 负责的引用类型，受 JVM 管理。

这么说可能有些难理解，仍旧以我们熟悉的 `android_media_MediaRecorder_native_init` 函数来举例，如下所示：

```
frameworks/base/media/jni/android_media_MediaRecorder.cpp
```

```
static void
android_media_MediaRecorder_native_init(JNIEnv *env)
{
    jclass clazz;
    clazz = env->FindClass("android/media/MediaRecorder");//1
    if (clazz == NULL) {
        return;
    }
    ...
}
```

注释 1 处的 `FindClass` 会返回 `clazz`，这个 `clazz` 就是本地引用，它会在 `android_media_MediaRecorder_native_init` 函数调用返回后被自动释放。我们也可以使用 JNIEnv 的 `DeleteLocalRef` 函数来手动删除本地引用，`DeleteLocalRef` 函数的使用场景主要是在 native 函数返回前占用了大量内存，需要调用 `DeleteLocalRef` 函数立即删除本地引用。

9.6.2 全局引用

全局引用和本地引用几乎是相反的，它主要有以下特点：

- 在 native 函数返回时不会被自动释放，因此全局引用需要手动来进行释放，并且不会被 GC 回收。
- 全局引用是可以跨线程使用的。
- 全局引用不受到 JVM 管理。

JNIEnv 的 NewGlobalRef 函数用来创建全局引用，调用 JNIEnv 的 DeleteGlobalRef 函数来释放全局引用：

```
frameworks/base/media/jni/android_media_MediaRecorder.cpp
```

```
JNIMediaRecorderListener::JNIMediaRecorderListener(JNIEnv* env, jobject thiz,
jobject weak_thiz)
{
    jclass clazz = env->GetObjectClass(thiz);//1
    if (clazz == NULL) {
        ALOGE("Can't find android/media/MediaRecorder");
        jniThrowException(env, "java/lang/Exception", NULL);
        return;
    }
    mClass = (jclass)env->NewGlobalRef(clazz);//2
    mObject = env->NewGlobalRef(weak_thiz);
}
```

在注释 1 处返回的 clazz 是本地引用，并传入到注释 2 处，在注释 2 处调用 JNIEnv 的 NewGlobalRef 函数将 clazz 转变为全局引用 mClass。那什么时候将全局引用 mClass 释放呢？我们来查看 JNIMediaRecorderListener 的析构函数：

```
frameworks/base/media/jni/android_media_MediaRecorder.cpp
```

```
JNIMediaRecorderListener::~JNIMediaRecorderListener()
{
    // remove global references
    JNIEnv *env = AndroidRuntime::getJNIEnv();
    env->DeleteGlobalRef(mObject);
    env->DeleteGlobalRef(mClass);//1
}
```

从上面的析构函数的注释就可以知道，这个析构函数用来释放全局引用，在注释 1 处释放了全局引用 mClass。

9.6.3 弱全局引用

弱全局引用是一种特殊的全局引用，它和全局引用的特点相似，不同的是弱全局是可

以被 GC 回收的，弱全局引用被 GC 回收之后会指向 NULL。JNIEnv 的 NewWeakGlobalRef 函数用来创建弱全局引用，调用 JNIEnv 的 DeleteWeakGlobalRef 函数来释放弱全局引用。由于弱全局引用可能被 GC 回收，因此在使用它之前要先判断它是否被回收了，方法就是使用 JNIEnv 的 IsSameObject 函数来判断：

```
if (env->IsSameObject(weakGlobalRef, NULL)) { //1
    return false;
}
...使用 weakGlobalRef
```

weakGlobalRef 是一个弱全局引用，在使用它之前，调用 JNIEnv 的 IsSameObject 函数，这个函数会判断传入的两个引用是否相等，如果相等返回 JNI_TRUE，不相等返回 JNI_FALSE。注释 1 处如果返回 JNI_TRUE 说明 weakGlobalRef 等于 NULL 并被 GC 回收了，就会调用 return false 不执行后面的代码逻辑，否则就使用弱全局引用 weakGlobalRef。

9.7 本章小结

本章从系统的源码的角度讲解了 JNI 的原理，包括 MediaRecorder 框架中的 JNI 的应用、数据类型的转换、方法签名、JNIEnv 和引用类型，作为应用开发了解这些知识点就足够了，如果想要了解更多的 JNI 的知识就需要阅读其他专门讲解 JNI 的书籍。

第 10 章

Java 虚拟机

关联章节：第 11 章 Dalvik 和 ART；第 12 章 理解 ClassLoader；第 16 章 绘制优化

有的读者会有疑问，介绍 Java 虚拟机的书很多啊！为何要在本书加入这一章呢？主要有两方面原因：

- 虽然有专门讲 Java 虚拟机的书，但是对于 Android 开发并不算友好，会看得比较吃力，作为 Android 开发并不需要了解那么多的知识点，因此这一章告诉你作为 Android 开发了解本章的内容基本就足够了，如果要深入了解则需要看专门讲 Java 虚拟机的书籍。
- 如果不理解 Java 虚拟机，就不利于这些章节的知识吸收。

10.1 概述

我们常说的 JDK (Java Development Kit) 包含了 Java 语言、Java 虚拟机和 Java API 类库这三部分，是 Java 程序开发的最小环境。而 JRE (Java Runtime Environment) 包含了 Java API 中的 Java SE API 子集和 Java 虚拟机这两部分，是 Java 程序运行的标准环境。那么可以看出 Java 虚拟机的重要性，它是整个 Java 平台的基石，是 Java 语言编译代码的运行平台。你可以把 Java 虚拟机看作一个抽象的计算机，它有各种指令集和各种运行时数据区域。虽然叫 Java 虚拟机，但其实在它之上运行的语言可不仅仅是 Java，还包括 Kotlin、Groovy、

Scala、Jython 等。因此对于 Android 开发来说，不管你开发用的是 Java 还是 Kotlin，你都需要去理解 Java 虚拟机。

10.1.1 Java虚拟机家族

有些读者可能认为 Java 虚拟机就是“一个”虚拟机而已，它还有家族？或者认为 Java 虚拟机指的就是 Oracle 的 HotSpot 虚拟机，这里来简单介绍 Java 虚拟机家族，自从 1996 年 Sun 公司发布的 JDK1.0 中包含的 Sun Classic VM 到今天，出现和消亡了很多种虚拟机，我们这里只简单介绍目前存活的相对主流 Java 虚拟机。

1. HotSpot VM

Oracle JDK 和 OpenJDK 中自带的虚拟机，是最主流的使用范围最广的 Java 虚拟机。介绍 Java 虚拟机的技术文章，如果不做特殊说明，大部分都是介绍 HotSpot VM 的。HotSpot VM 并非是 Sun 公司开发的，而是由 Longview Technologies 这家小公司设计的，它在 1997 年被 Sun 公司收购，Sun 公司又在 2009 年被 Oracle 收购。

2. J9 VM

J9 VM 是 IBM 开发的虚拟机，目前是其主力发展的 Java 虚拟机。J9 VM 的市场定位和 HotSpot VM 接近，它是一款设计上从服务器端到桌面应用再到嵌入式都考虑到的多用途虚拟机，目前 J9 VM 的性能水平大致与 HotSpot VM 是一个档次的。

3. Zing VM

以 Oracle 的 HotSpot VM 为基础，改进了许多影响延迟的细节。最大的 3 个卖点如下：

- 低延迟，“无暂停”的 C4 GC，GC 带来的暂停可以控制在 10ms 以下的级别，支持的 Java 堆大小可以达到 1TB。
- 启动后快速预热功能。
- 可管理性：零开销、可在生产环境全时开启、整合在 JVM 内的监控工具 Zing Vision。

需要注意的是，Android 中的 Dalvik 和 ART 虚拟机并不属于 Java 虚拟机，因此这里没有列出它们，关于 Dalvik 和 ART 虚拟机将在第 11 章进行介绍。

10.1.2 Java虚拟机执行流程

当我们执行一个 Java 程序时，它的执行流程如图 10-1 所示。

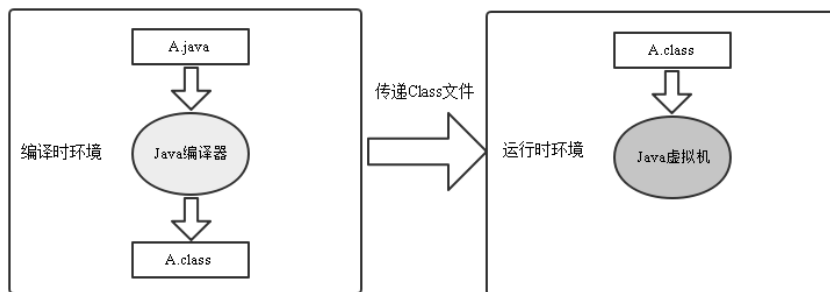


图 10-1 Java 虚拟机执行流程

从图 10-1 中可以发现 Java 虚拟机执行流程分为两大部分,分别是编译时环境和运行时环境,当一个 Java 文件经过 Java 编译器编译后会生成 Class 文件,这个 Class 文件会由 Java 虚拟机来进行处理。Java 虚拟机与 Java 语言没有什么必然的联系,它只与特定的二进制文件:Class 文件有关。因此无论任何语言只要能编译成 Class 文件,就可以被 Java 虚拟机识别并执行,如图 10-2 所示。

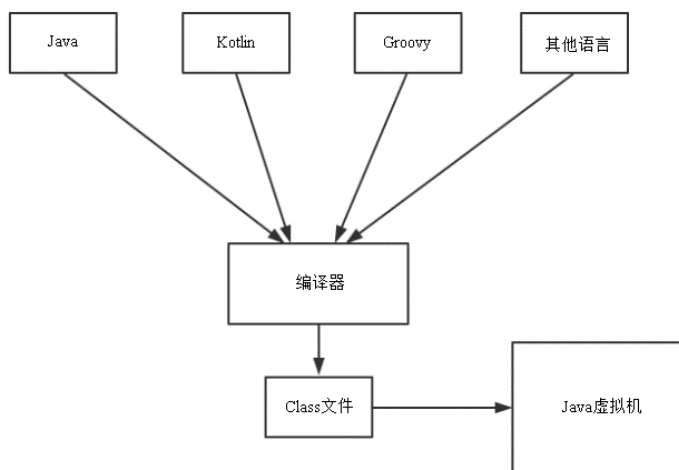


图 10-2 语言与 Java 虚拟机

10.2 Java虚拟机结构

这里所讲的体系结构,指的是 Java 虚拟机的抽象行为,而不是具体的比如 HotSpot VM 的实现。按照 Java 虚拟机规范,抽象的 Java 虚拟机如图 10-3 所示。

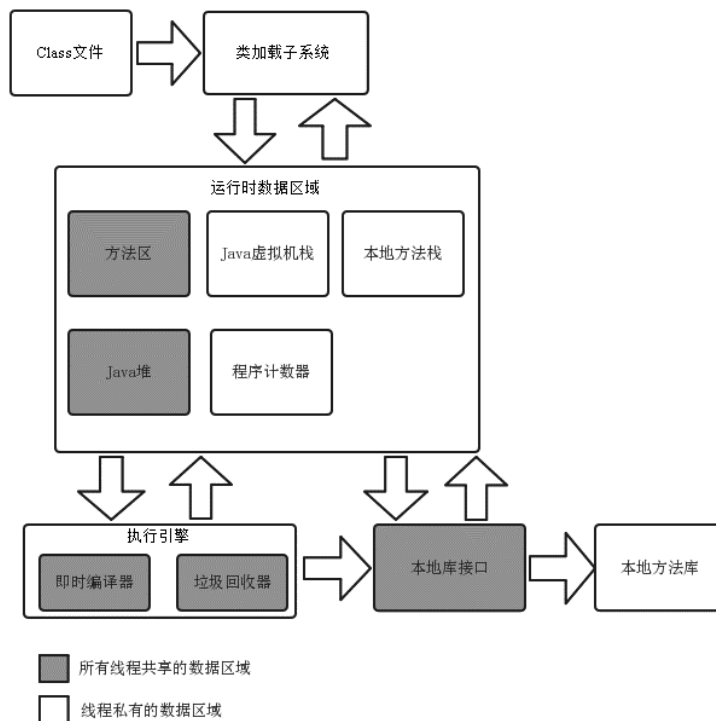


图 10-3 Java 虚拟机结构

从图 10-3 可以看出 Java 虚拟机结构包括运行时数据区域、执行引擎、本地库接口和本地方法库，其中类加载子系统并不属于 Java 虚拟机的内部结构。图 10-3 中标出了线程共享和线程私有的区域，比如方法区和 Java 堆就是所有线程共享的数据区域。下面针对图 10-3 来介绍 Android 开发需要掌握的 Class 文件格式和运行时数据区域。

10.2.1 Class文件格式

Java 文件被编译后生成了 Class 文件，这种二进制格式文件不依赖于特定的硬件和操作系统。每一个 Class 文件中都对对应着唯一的类或者接口的定义信息，但是类或者接口并不一定定义在文件中，比如类和接口可以通过类加载器来直接生成。10.1.2 节中我们知道无论任何语言只要能编译成 Class 文件，就可以被 Java 虚拟机识别并执行，可见 Class 文件的重要性，了解它对于我们学习那些基于 Java 虚拟机的语言会有很大帮助。下面我们来学习 Class 文件格式，如下所示：

```

ClassFile {
    u4 magic;//魔数, 固定值为 0xCAFEFABE, 用来判断当前文件是不是能被 Java 虚拟机处理的 Class 文件
    u2 minor_version;                //副版本号
    u2 major_version;                //主版本号
    u2 constant_pool_count;           //常量池计数器
    cp_info constant_pool[constant_pool_count-1]; //常量池
    u2 access_flags;                 //类和接口层次的访问标志
    u2 this_class;                   //类索引
    u2 super_class;                  //父类索引
    u2 interfaces_count;             //接口计数器
    u2 interfaces[interfaces_count]; //接口表
    u2 fields_count;                 //字段计数器
    field_info fields[fields_count]; //字段表
    u2 methods_count;               //方法计数器
    method_info methods[methods_count]; //方法表
    u2 attributes_count;            //属性计数器
    attribute_info attributes[attributes_count]; //属性表
}

```

可以看到 ClassFile 具有很强的描述能力, 包含了很多关键的信息, 其中 u4、u2 表示“基本数据类型”, class 文件的基本数据类型如下所示。

- u1: 1 字节, 无符号类型。
- u2: 2 字节, 无符号类型。
- u4: 4 字节, 无符号类型。
- u8: 8 字节, 无符号类型。

10.2.2 类的生命周期

一个 Java 文件被加载到 Java 虚拟机内存中到从内存中卸载的过程被称为类的生命周期。类的生命周期包括的阶段分别是: 加载、链接、初始化、使用和卸载, 其中链接包括了三个阶段: 验证、准备和解析, 因此类的生命周期包括了 7 个阶段。广义上来说类的加载包括了类的生命周期的 5 个阶段, 分别是加载、链接 (验证、准备和解析)、初始化。如图 10-4 所示。

接下来大概介绍类的加载各个阶段所做的工作, 如下所示。

- (1) 加载: 查找并加载 Class 文件。
- (2) 链接: 包括验证、准备和解析。

- 验证：确保被导入类型的正确性。
- 准备：为类的静态字段分配字段，并用默认值初始化这些字段。
- 解析：虚拟机将常量池内的符号引用替换为直接引用。

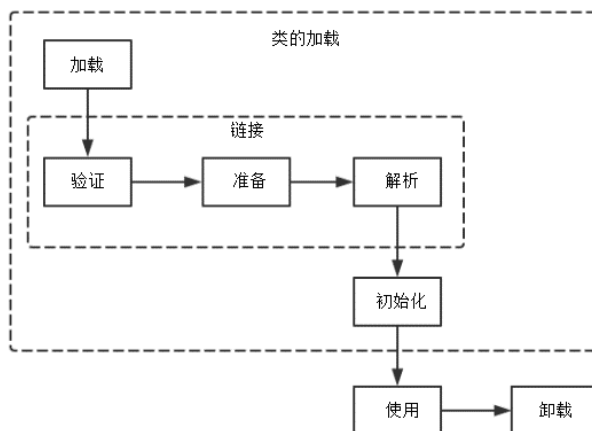


图 10-4 类的生命周期

(3) 初始化：将类变量初始化为正确初始值。

根据《深入理解 Java 虚拟机》的描述，加载阶段（不是类的加载）主要做了 3 件事情：

- 根据特定名称查找类或接口类型的二进制字节流。
- 将这个二进制字节流所代表的静态存储结构转化为方法区的运行时数据结构。
- 在内存中生成一个代表这个类的 `java.lang.Class` 对象，作为方法区这个类的各种数据的访问入口。

其中第一件事情就是由 Java 虚拟机外部的类加载子系统来完成的，下面我们来学习类加载子系统。

10.2.3 类加载子系统

类加载子系统通过多种类加载器来查找和加载 Class 文件到 Java 虚拟机中，Java 虚拟机有两种类加载器：系统加载器和自定义加载器。其中系统加载器包括以下三种。

1. Bootstrap ClassLoader（引导类加载器）

用 C/C++ 代码实现的加载器，用于加载指定的 JDK 的核心类库，比如 `java.lang.`、`java.util.` 等这些系统类。它用来加载以下目录中的类库：

- \$JAVA_HOME/jre/lib 目录。
- -Xbootclasspath 参数指定的目录。

Java 虚拟机的启动就是通过引导类加载器创建一个初始类来完成的。由于类加载器是使用平台相关的底层 C/C++语言实现的，所以该加载器不能被 Java 代码访问到，但是我们可以查询某个类是否被引导类加载器加载过。

2. Extensions ClassLoader（拓展类加载器）

用于加载 Java 的拓展类，提供除了系统类之外的额外功能。它用来加载以下目录中的类库：

- 加载\$JAVA_HOME/jre/lib/ext 目录。
- 系统属性 java.ext.dir 所指定的目录。

3. Application ClassLoader（应用程序类加载器）

又称作 System ClassLoader（系统类加载器），这是因为这个类加载器可以通过 ClassLoader 的 `getSystemClassLoader` 方法获取到。它用来加载以下目录中的类库：

- 当前应用程序 Classpath 目录。
- 系统属性 java.class.path 指定的目录。

除了系统加载器还有自定义加载器，它是通过继承 `java.lang.ClassLoader` 类的方式来实现自己的类加载器的。关于类加载器这里只是简单介绍，在 12 章会进行详细介绍。

10.2.4 运行时数据区域

很多人将 Java 的内存分为堆内存（Heap）和栈内存（Stack），这种分法不够准确，Java 的内存区域划分实际上远比这要复杂。Java 虚拟机在执行 Java 程序的过程中会把它所管理的内存划分为不同的数据区域，根据《Java 虚拟机规范（Java SE7 版）》的规定，这些数据区域分别为程序计数器、Java 虚拟机栈、本地方法栈、Java 堆和方法区，下面一一对它们进行介绍。

1. 程序计数器

为了保证程序能够连续地执行下去，处理器必须具有某些手段来确定下一条指令的地址，而程序计数器正是起到这种作用。程序计数器（Program Counter Register）也叫作 PC 寄存器，是一块较小的内存空间。在虚拟机概念模型中，字节码解释器工作时就是通过改

变程序计数器来选取下一条需要执行的字节码指令的，Java 虚拟机的多线程是通过轮流切换并分配处理器执行时间的方式来实现的，在一个确定的时刻只有一个处理器执行一条线程中的指令，为了在线程切换后能恢复到正确的执行位置，每个线程都会有一个独立的程序计数器，因此，程序计数器是线程私有的。如果线程执行的方法不是 Native 方法，则程序计数器保存正在执行的字节码指令地址，如果是 Native 方法则程序计数器的值为空 (Undefined)。程序计数器是 Java 虚拟机规范中唯一没有规定任何 OutOfMemoryError 情况的数据区域。

2. Java 虚拟机栈

每一条 Java 虚拟机线程都有一个线程私有的 Java 虚拟机栈 (Java Virtual Machine Stacks)。它的生命周期与线程相同，与线程是同时创建的。Java 虚拟机栈存储线程中 Java 方法调用的状态，包括局部变量、参数、返回值以及运算的中间结果等。一个 Java 虚拟机栈包含了多个栈帧，一个栈帧用来存储局部变量表、操作数栈、动态链接、方法出口等信息。当线程调用一个 Java 方法时，虚拟机压入一个新的栈帧到该线程的 Java 虚拟机栈中，在该方法执行完成后，这个栈帧就从 Java 虚拟机栈中弹出。我们平常所说的栈内存 (Stack) 指的就是 Java 虚拟机栈。Java 虚拟机规范中定义了两种异常情况。

- 如果线程请求分配的栈容量超过 Java 虚拟机所允许的最大容量，Java 虚拟机会抛出 StackOverflowError。
- 如果 Java 虚拟机栈可以动态扩展（大部分 Java 虚拟机都可以动态扩展），但是扩展时无法申请到足够的内存，或者在创建新的线程时没有足够的内存去创建对应的 Java 虚拟机栈，则会抛出 OutOfMemoryError 异常。

3. 本地方法栈

Java 虚拟机实现可能要用到 C Stacks 来支持 Native 语言，这个 C Stacks 就是本地方法栈 (Native Method Stack)。它与 Java 虚拟机栈类似，只不过本地方法栈是用来支持 Native 方法的。如果 Java 虚拟机不支持 Native 方法，并且也不依赖于 C Stacks，可以无须支持本地方法栈。在 Java 虚拟机规范中对本地方法栈的语言和数据结构等没有强制规定，因此具体的 Java 虚拟机可以自由实现它，比如 HotSpot VM 将本地方法栈和 Java 虚拟机栈合二为一。与 Java 虚拟机栈类似，本地方法栈也会抛出 StackOverflowError 和 OutOfMemoryError 异常。

4. Java 堆

Java 堆 (Java Heap) 是被所有线程共享的运行时内存区域。Java 堆用来存放对象实例，

几乎所有的对象实例都在这里分配内存。Java 堆存储的对象被垃圾收集器管理，这些受管理的对象无法显式地销毁。从内存回收的角度来分，Java 堆可以粗略地分为新生代和老年代，从内存分配的角度 Java 堆中可能划分出多个线程私有的分配缓冲区。不管如何划分，Java 堆存储的内容是不变的，进行划分是为了能更快地回收或者分配内存。Java 堆的容量可以是固定的，也可以动态扩展。Java 堆所使用的内存在物理上不需要连续，逻辑上连续即可。Java 虚拟机规范中定义了一种异常情况：如果在堆中没有足够的内存来完成实例分配，并且堆也无法进行扩展时，则会抛出 `OutOfMemoryError` 异常。

5. 方法区

方法区（Method Area）是被所有线程共享的运行时内存区域，用来存储已经被 Java 虚拟机加载的类的结构信息，包括运行时常量池、字段和方法信息、静态变量等数据。方法区是 Java 堆的逻辑组成部分，它一样在物理上不需要连续，并且可以选择在方法区中不实现垃圾收集。方法区并不等同于永久代，只是因为 HotSpot VM 使用永久代来实现方法区，对于其他的 Java 虚拟机，比如 J9 和 JRockit 等，并不存在永久代概念。

在 Java 虚拟机规范中定义了一种异常情况：如果方法区的内存空间不满足内存分配需求时，Java 虚拟机会抛出 `OutOfMemoryError` 异常。

6. 运行时常量池

运行时常量池（Runtime Constant Pool）并不是运行时数据区域的其中一份子，而是方法区的一部分。在 10.2.1 节中我们得知，Class 文件不仅包含类的版本、接口、字段和方法等信息，还包含常量池，它用来存放编译时期生成的字面量和符号引用，这些内容会在类加载后存放在方法区的运行时常量池中。运行时常量池可以理解为是类或接口的常量池的运行时表现形式。

在 Java 虚拟机规范中定义了一种异常情况：当创建类或接口时，如果构造运行时常量池所需的内存超过了方法区所能提供的最大值，Java 虚拟机会抛出 `OutOfMemoryError` 异常。

10.3 对象的创建

对象的创建是我们经常要做的事，通常是通过 `new` 指令来完成一个对象的创建的，当虚拟机接收到一个 `new` 指令时，它会做如下的操作。

1. 判断对象对应的类是否加载、链接和初始化

虚拟机接收到一条 new 指令时，首先会去检查这个指定的参数是否能在常量池中定位到一个类的符号引用，并且检查这个符号引用代表的类是否已被类加载器加载、链接和初始化过。

2. 为对象分配内存

类加载完成后，接着会在 Java 堆中划分一块内存分配给对象。内存分配根据 Java 堆是否规整，有两种方式。

- 指针碰撞：如果 Java 堆的内存是规整的，即所有用过的内存放在一边，而空闲的内存放在另一边。分配内存时将位于中间的指针指示器向空闲的内存移动一段与对象大小相等的距离，这样便完成分配内存工作。
- 空闲列表：如果 Java 堆的内存不是规整的，则需要由虚拟机维护一个列表来记录哪些内存是可用的，这样在分配的时候可以从列表中查询到足够大的内存分配给对象，并在分配后更新列表记录。

Java 堆的内存是否规整根据所采用的垃圾收集器是否带有压缩整理功能有关。

3. 处理并发安全问题

创建对象是一个非常频繁的操作，所以需要解决并发的问题，有两种方式：

- 对分配内存空间的动作进行同步处理，比如在虚拟机采用 CAS 算法并配上失败重试的方式保证更新操作的原子性。
- 每个线程在 Java 堆中预先分配一小块内存，这块内存称为本地线程分配缓冲 (Thread Local Allocation Buffer, TLAB)，线程需要分配内存时，就在对应线程的 TLAB 上分配内存，当线程中的 TLAB 用完并且被分配到了新的 TLAB 时，这时候才需要同步锁定。通过 -XX:+/-UserTLAB 参数来设定虚拟机是否使用 TLAB。

4. 初始化分配到的内存空间

将分配到的内存，除了对象头外都初始化为零值。

5. 设置对象的对象头

将对象的所属类、对象的 hashCode 和对象的 GC 分代年龄等数据存储在对象的对象头中。关于对象头，10.4 节中会进行介绍。

6. 执行 init 方法进行初始化

执行 init 方法，初始化对象的成员变量、调用类的构造方法，这样一个对象就被创建了出来。

10.4 对象的堆内存布局

对象创建完毕，并且已经在 Java 堆中分配了内存，那么对象在堆内存是如何进行布局的呢？以 HotSpot 虚拟机为例，对象在堆内存的布局分为三个区域，分别是对象头(Header)、实例数据 (Instance Data)、对齐填充 (Padding)。下面分别来对这三个区域进行简单介绍。

- 对象头：对象头包括两部分信息，分别是 Mark Word 和元数据指针，Mark Word 用于存储对象运行时的数据，比如 hashCode、锁状态标志、GC 分代年龄、线程持有的锁等。而元数据指针用于指向方法区中的目标类的元数据，通过元数据可以确定对象的具体类型，具体是如何实现的请看 10.5 节。
- 实例数据：用于存储对象中的各种类型的字段信息（包括从父类继承来的）。
- 对齐填充：对齐填充不一定存在，起到了占位符的作用，没有特别的含义。

Mark Word 在 HotSpot 中的实现类为 markOop.hpp，markOop 被设计成一个非固定的数据结构，这是为了在极小的空间中存储尽量多的数据，32 位虚拟机的 markOop 格式如下所示：

```
hash:25 ----->| age:4    biased_lock:1 lock:2 (normal object)
JavaThread*:23 epoch:2 age:4    biased_lock:1 lock:2 (biased object)
size:32 ----->| (CMS free block)
PromotedObject*:29 ----->| promo_bits:3 ----->| (CMS promoted object)
```

其中部分数据类型解释如下。

- hash：对象的哈希码。
- age：对象的分代年龄。
- biased_lock：偏向锁标识位。
- lock：锁状态标识位。
- JavaThread*：持有偏向锁的线程 ID。
- epoch：偏向时间戳。

对象的堆内存布局如图 10-5 所示。



图 10-5 对象的堆内存布局

10.5 oop-class模型

oop-class 模型是用来描述 Java 对象实例的一种模型，它分为两个部分，OOP (Ordinary Object Pointer) 指的是普通对象指针，用来表示对象的实例信息。class 用来描述元数据。

在 HotSpot 中就采用了 oop-class 模型，oop 实际是一个家族，Java 虚拟机内部会定义很多 oop 类型，如下所示：

```
hotspot/src/share/vm/oops/oopsHierarchy.hpp

typedef class  markOopDesc*           markOop; //oop 标记对象
typedef class  oopDesc*               oop; //oop 家族的顶级父类
typedef class  instanceOopDesc*       instanceOop; //表示 Java 类实例
typedef class  arrayOopDesc*          arrayOop; //数组对象
typedef class  objArrayOopDesc*       objArrayOop; //引用类型数组对象
typedef class  typeArrayOopDesc*      typeArrayOop; //基本类型数组对象
```

其中 oopDesc 是所有 oop 的顶级父类，arrayOopDesc 是 objArrayOopDesc 和 typeArrayOopDesc 的父类。instanceOopDesc 和 arrayOopDesc 都可以用来描述对象头。

在 oopsHierarchy.hpp 中还定义了 klass 家族：

```
hotspot/src/share/vm/oops/oopsHierarchy.hpp

class  Klass; //klass 家族的父类
class  InstanceKlass; //描述 Java 类的数据结构
class  InstanceMirrorKlass; //描述 java.lang.Class 实例
class  InstanceClassLoaderKlass; //特殊的 InstanceKlass，不添加任何字段
class  InstanceRefKlass; //描述 java.lang.ref.Reference 的子类
class  ArrayKlass; //描述 Java 数组信息
class  ObjArrayKlass; //描述 Java 中引用类型数组的数据结构
class  TypeArrayKlass; //描述 Java 中基本类型数组的数据结构
```

其中 Klass 是 klass 家族的父类（不是顶级父类），ArrayKlass 是 ObjArrayKlass 和 TypeArrayKlass 的父类，可以发现 oop 家族的成员和 klass 家族的成员有着对应的关系，比如 instanceOopDesc 对应 InstanceKlass，objArrayOopDesc 对应 ObjArrayKlass。这里我们拿 instanceOopDesc 和 InstanceKlass 的对应关系来举例，其他的对应关系也是类似的。instanceOopDesc 的定义如下所示：

```
hotspot/src/share/vm/oops/instanceOop.hpp
```

```
class instanceOopDesc : public oopDesc {
public:
    static int header_size() { return sizeof(instanceOopDesc)/HeapWordSize; }
    static int base_offset_in_bytes() {
        return (UseCompressedOops && UseCompressedClassPointers) ?
            klass_gap_offset_in_bytes() :
            sizeof(instanceOopDesc);
    }
    static bool contains_field_offset(int offset, int nonstatic_field_size) {
        int base_in_bytes = base_offset_in_bytes();
        return (offset >= base_in_bytes &&
            (offset-base_in_bytes) < nonstatic_field_size * heapOopSize);
    }
};
```

可以看出 instanceOopDesc 继承了 oopDesc：

```
openjdk/hotspot/src/share/vm/oops/oop.hpp
```

```
class oopDesc {
    friend class VMStructs;
private:
    volatile markOop _mark;
    union _metadata {
        Klass* _klass;
        narrowKlass _compressed_klass;
    } _metadata;
    // Fast access to barrier set. Must be initialized.
    static BarrierSet* _bs;
    ...
};
```

oopDesc 中包含两个数据成员：mark 和 _metadata。其中 markOop 类型的 mark 对象指的是前面讲到的 Mark World。metadata 是一个共用体，其中 klass 是普通指针，_compressed_klass 是压缩类指针，它们就是 10.4 节讲到的元数据指针，这两个指针根据对应关系都会指向

instanceKlass, instanceKlass 可以用来描述元数据, 我们接着往下看, instanceKlass 的代码如下所示:

openjdk/hotspot/src/share/vm/oops/instanceKlass.hpp

```
class InstanceKlass: public Klass {
    ...
    enum ClassState {
        allocated,
        loaded,
        linked,
        being_initialized,
        fully_initialized,
        initialization_error
    };
    ...
}
```

instanceKlass 继承自 klass, 枚举 ClassState 用来标识对象的加载进度, klass 中定义的部分字段如下所示:

hotspot/src/share/vm/oops/klass.hpp

```
jint      _layout_helper; //对象布局的综合描述符
Symbol*   _name; //类名
oop       _java_mirror; //类的镜像类
Klass*    _super; //父类
Klass*    _subklass; //第一个子类
Klass*    _next_sibling; //下一个兄弟节点
jint      _modifier_flags; //修饰符标识
AccessFlags _access_flags; //访问权限标识
```

可以看到 klass 描述了元数据, 具体来说就是 Java 类在 Java 虚拟机中对等的 C++ 类型描述, 这样继承自 klass 的 instanceKlass 同样可以用来描述元数据。了解了 opp-klass 模型, 我们就可以分析 Java 虚拟机是如何通过栈帧中的对象引用找到对应的对象实例的, 如图 10-6 所示。

从图 10-6 中可以看出, Java 虚拟机通过栈帧中的对象引用找到 Java 堆中的 instanceOopDesc, 这样就可以访问到 Java 对象的实例信息, 当需要访问对象的具体类型等信息时, 可以通过 instanceOopDesc 中的元数据指针来找到方法区中对应的 instanceKlass。

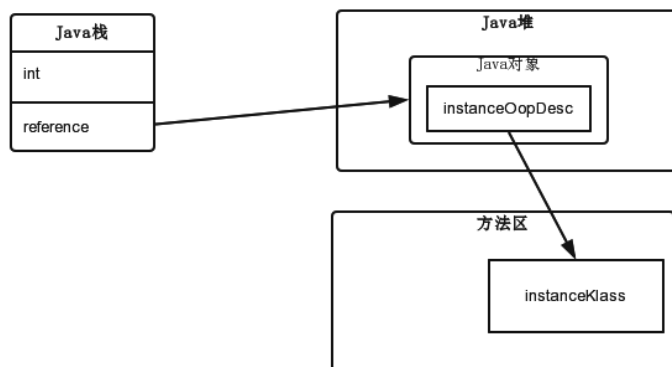


图 10-6 确定对象具体类型

10.6 垃圾标记算法

垃圾收集器 (Garbage Collection)，通常被称作 GC。提到 GC，很多人认为它是伴随 Java 而出现的，其实 GC 出现的时间要比 Java 早太多了，它是 1960 年诞生于 MIT 的 Lisp。GC 主要做了两个工作，一个是内存的划分和分配，另一个是对垃圾进行回收。关于内存的划分和分配，目前 Java 虚拟机内存的划分是依赖于 GC 设计的，比如现在 GC 都是采用了分代收集算法来回收垃圾的，Java 堆作为 GC 主要管理的区域，被细分为新生代和老年代，再细致一点新生代又可以划分为 Eden 空间、From Survivor 空间、To Survivor 空间等，这样划分是为了更快地进行内存分配和回收。空间划分后，GC 就可以为新对象分配内存空间。关于对垃圾进行回收，被引用的对象是存活的对象，而不被引用的对象是死亡的对象（也就是垃圾），GC 要区分出存活的对象和死亡的对象（也就是垃圾标记），并对垃圾进行回收。在对垃圾进行回收前，GC 要先标记出垃圾，那么如何标记呢？目前有两种垃圾标记算法，分别是引用计数算法和根搜索算法，这两个算法都和引用有些关联，因此讲垃圾标记算法前，我们先回顾一下引用的知识点。

10.6.1 Java中的引用

在 JDK1.2 之后，Java 将引用分为强引用、软引用、弱引用和虚引用。

1. 强引用

当我们新建一个对象时就创建了一个具有强引用的对象，如果一个对象具有强引用，

垃圾收集器就绝不会回收它。Java 虚拟机宁愿抛出 `OutOfMemoryError` 异常，使程序异常终止，也不会回收具有强引用的对象来解决内存不足的问题。

2. 软引用

如果一个对象只具有软引用，当内存不够时，会回收这些对象的内存，回收后如果还是没有足够的内存，就会抛出 `OutOfMemoryError` 异常。Java 提供了 `SoftReference` 类来实现软引用。

3. 弱引用

弱引用比起软引用具有更短的生命周期，垃圾收集器一旦发现了只具有弱引用的对象，不管当前内存是否足够，都会回收它的内存。Java 提供了 `WeakReference` 类来实现弱引用。

4. 虚引用

虚引用并不会决定对象的生命周期，如果一个对象仅持有虚引用，这就和没有任何引用一样，在任何时候都可能被垃圾收集器回收。一个只具有虚引用的对象，被垃圾收集器回收时会收到一个系统通知，这也是虚引用的主要作用。Java 提供了 `PhantomReference` 类来实现虚引用。

10.6.2 引用计数算法

引用计数算法的基本思想就是每个对象都有一个引用计数器，当对象在某处被引用的时候，它的引用计数器就加 1，引用失效时就减 1。当引用计数器中的值变为 0，则该对象就不能被使用，变成了垃圾。

目前主流的 Java 虚拟机没有选择引用计数算法来为垃圾标记，主要原因是引用计数算法没有解决对象之间相互循环引用的问题。

举个例子，在下面代码的注释 1 和注释 2 处，`d1` 和 `d2` 相互引用，除此之外这两个对象无任何其他引用，实际上这两个对象已经死亡，应该作为垃圾被回收，但是由于这两个对象互相引用，引用计数就不会为 0，如果 Java 虚拟机采用了引用计数算法，垃圾收集器就无法回收它们。

```
class _2MB_Data {  
    public Object instance = null;  
    private byte[] data = new byte[2 * 1024 * 1024]; //用来占内存，测试垃圾回收  
}
```

```
public class ReferenceGC {
    public static void main(String[] args) {
        _2MB_Data d1 = new _2MB_Data();
        _2MB_Data d2 = new _2MB_Data();
        d1.instance = d2;//1
        d2.instance = d1;//2
        d1 = null;
        d2 = null;
        System.gc();
    }
}
```

如果你使用 Android Studio，就在 Edit Configurations 中的 VM options 加入如下语句来输出详细的 GC 日志：

```
-XX:+PrintGCDetails
```

运行程序，GC 日志为：

```
GC (System.gc()) [PSYoungGen: 8028K->832K(76288K)] 8028K->840K(251392K), 0.0078334
secs Full GC (System.gc()) [PSYoungGen: 832K->0K(76288K)] [ParOldGen: 8K->603K
(175104K)] 840K->603K(251392K), [Metaspace: 3015K->3015K(1056768K)], 0.0045844 secs
Heap
  PSYoungGen total 76288K, used 1966K [0x000000076af80000, 0x0000000770480000,
0x000000007c0000000)
    eden space 65536K, 3% used [0x000000076af80000,0x000000076b16bac0,0x000000076
ef80000)
    from space 10752K, 0% used [0x000000076ef80000,0x000000076ef80000,0x000000076
fa00000)
    to space 10752K, 0% used [0x000000076fa00000,0x000000076fa00000,0x0000000770
480000)
  ParOldGen total 175104K, used 603K [0x000000006c0e0000, 0x000000006cb90000,
0x0000000076af80000)
    object space 175104K, 0% used [0x000000006c0e0000,0x000000006c0e96d10,0x000000006
cb900000)
  Metaspace used 3046K, capacity 4496K, committed 4864K, reserved 1056768K class space
used 334K, capacity 388K, committed 512K, reserved 1048576K
```

查看此 GC 日志前我们先来简单了解一下各参数的含义，[GC (System.gc())]和[Full GC (System.gc())]说明了这次垃圾收集的停顿类型，而不是来区分新生代 GC 和老年代 GC 的。[Full GC (System.gc())]说明这次 GC 发生了 STW，STW 也就是 Stop the World 机制，意思是说在执行垃圾收集算法时，只有 GC 线程在运行，其他的线程则会全部暂停，等待 GC 线程执行完毕后才能再次运行。

PSYoungGen 代表新生代，ParOldGen 代表老年代，Metaspace 代表元空间（JDK 8 中用来替代永久代 PermGen）。

我们来看日志的[GC (System.gc())，内存变化为：8028K→840K(251392K)，8028K 代表回收前的内存大小，840K 代表回收后的内存大小，251392K 代表内存总大小。因此可以得知内存回收大小为（8028-840）KB。这就说明 JDK 8 的 HotSpot 虚拟机并没有引用计数算法来标记内存，它对上述代码中的两个死亡对象的引用进行了回收。

10.6.3 根搜索算法

这个算法的基本思想就是选定一些对象作为 GC Roots，并组成根对象集合，然后以这些 GC Roots 的对象作为起始点，向下搜索，如果目标对象到 GC Roots 是连接着的，我们则称该目标对象是可达的，如果目标对象不可达则说明目标对象是可以被回收的对象，如图 10-7 所示。

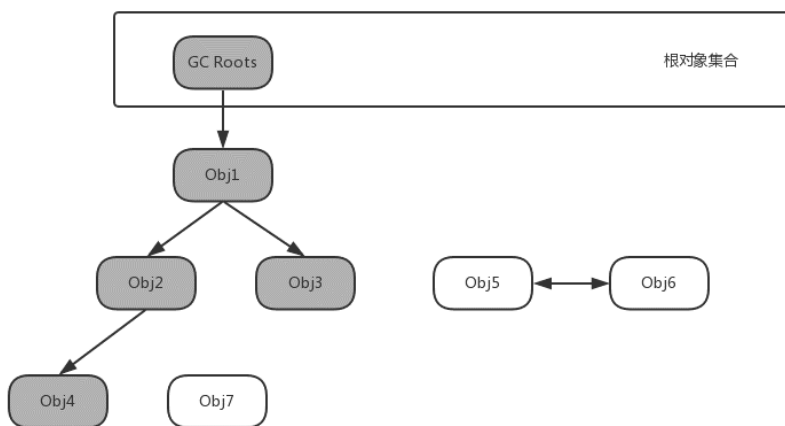


图 10-7 根对象集合

从图 10-7 可以看出，Obj5、Obj6 和 Obj7 都是不可达的对象，其中 Obj5 和 Obj6 虽然互相引用，但是因为它们到 GC Roots 是不可达的，所以它们仍旧被判定为可回收的对象，这样根搜索算法就解决了引用计数算法无法解决的问题：已经死亡的对象因为相互引用而不能被回收。在 Java 中，可以作为 GC Roots 的对象主要有以下几种：

- Java 栈中引用的对象。
- 本地方法栈中 JNI 引用的对象。
- 方法区中运行时常量池引用的对象。

- 方法区中静态属性引用的对象。
- 运行中的线程。
- 由引导类加载器加载的对象。
- GC 控制的对象。

还有一个问题是被标记为不可达的对象会立即被垃圾收集器回收吗？要回答这个问题我们首先要了解 Java 对象在虚拟机中的生命周期。

10.7 Java对象在虚拟机中的生命周期

在 Java 对象被类加载器加载到虚拟机中后，Java 对象在 Java 虚拟机中有 7 个阶段。

1. 创建阶段（Created）

创建阶段的具体步骤为：

- (1) 为对象分配存储空间。
- (2) 构造对象。
- (3) 从超类到子类对 static 成员进行初始化。
- (4) 递归调用超类的构造方法。
- (5) 调用子类的构造方法。

2. 应用阶段（In Use）

当对象被创建，并分配给变量赋值时，状态就切换到了应用阶段。这一阶段的对象至少要具有一个强引用，或者显式地使用软引用、弱引用或者虚引用。

3. 不可见阶段（Invisible）

在程序中找不到对象的任何强引用，比如程序的执行已经超出了该对象的作用域。在不可见阶段，对象仍可能被特殊的强引用 GC Roots 持有着，比如对象被本地方法栈中 JNI 引用或被运行中的线程引用等。

4. 不可达阶段 (Unreachable)

在程序中找不到对象的任何强引用，并且垃圾收集器发现对象不可达。

5. 收集阶段 (Collected)

垃圾收集器已经发现对象不可达，并且垃圾收集器已经准备好要对该对象的内存空间重新进行分配，这个时候如果该对象重写了 `finalize` 方法，则会调用该方法。

6. 终结阶段 (Finalized)

在对象执行完 `finalize` 方法后仍然处于不可达状态时，或者对象没有重写 `finalize` 方法，则该对象进入终结阶段，并等待垃圾收集器回收该对象空间。

7. 对象空间重新分配阶段 (Deallocated)

当垃圾收集器对对象的内存空间进行回收或者再分配时，这个对象就会彻底消失。

好了，我们已经了解了 Java 对象在虚拟机中的生命周期，再来回想 10.6.3 节说的`问题`：被标记为不可达的对象会立即被垃圾收集器回收吗？很显然是不会的，被标记为不可达的对象会进入收集阶段，这时会执行该对象重写的 `finalize` 方法，如果没有重写 `finalize` 方法或者 `finalize` 方法中没有重新与一个可达的对象进行关联才会进入终结阶段，并最终被回收。

10.8 垃圾收集算法

在 10.6 节中我们学习了垃圾标记算法，垃圾被标记后，GC 就会对垃圾进行收集，垃圾收集有很多种算法，这一节就来介绍常用的垃圾收集算法的思想。

10.8.1 标记—清除算法

标记—清除算法 (Mark-Sweep) 是一种常见的基础垃圾收集算法，它将垃圾收集分为两个阶段。

- 标记阶段：标记出可以回收的对象。
- 清除阶段：回收被标记的对象所占用的空间。

标记—清除算法之所以是基础的，是因为后面讲到的垃圾收集算法都是在此算法的基础上进行改进的。标记—清除算法的执行过程如图 10-8 所示。

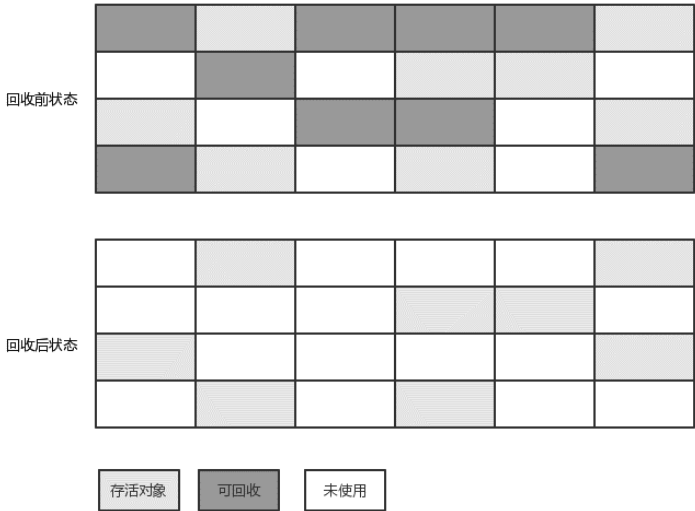


图 10-8 标记—清除算法的执行过程

标记—清除算法主要有两个缺点，一个是标记和清除的效率都不高，另一个从图 10-8 就可以看出来，就是容易产生大量不连续的内存碎片，碎片太多可能会导致后续没有足够的连续内存分配给较大的对象，从而提前触发新的一次垃圾收集动作。

10.8.2 复制算法

为了解决标记—清除算法的效率不高的问题，产生了复制算法。它把内存空间划为两个相等的区域，每次只使用其中一个区域。在垃圾收集时，遍历当前使用的区域，把存活对象复制到另外一个区域中，最后将当前使用的区域的可回收的对象进行回收。复制算法的执行过程如图 10-9 所示。

这种算法每次都对整个半区进行内存回收，不需要考虑内存碎片的问题，代价就是使用内存为原来的一半。复制算法的效率与存活对象的数目多少有很大的关系，如果存活对象很少，复制算法的效率就会很高。由于绝大多数对象的生命周期很短，并且这些生命周期很短的对象都存于新生代中，所以复制算法被广泛应用于新生代中，关于新生代中复制算法的应用，会在后面的分代收集算法中详细介绍。

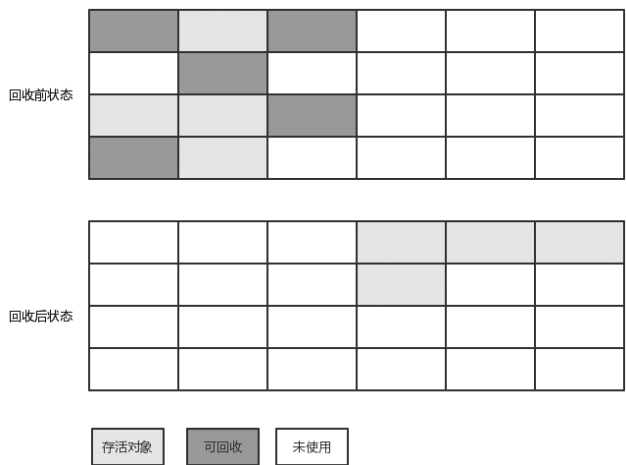


图 10-9 复制算法的执行过程

10.8.3 标记—压缩算法

在新生代中可以使用复制算法，但是在老年代就不能选择复制算法了，因为老年代的对象存活率会较高，这样会有较多的复制操作，导致效率变低。标记—清除算法可以应用在老年代中，但是它效率不高，在内存回收后容易产生大量内存碎片。因此就出现了一种标记—压缩（Mark-Compact）算法，与标记—清除算法不同的是，在标记可回收的对象后将所有存活的对象压缩到内存的一端，使它们紧凑地排列在一起，然后对边界以外的内存进行回收，回收后，已用和未用的内存都各自一边，标记—压缩算法的执行过程如图 10-10 所示。

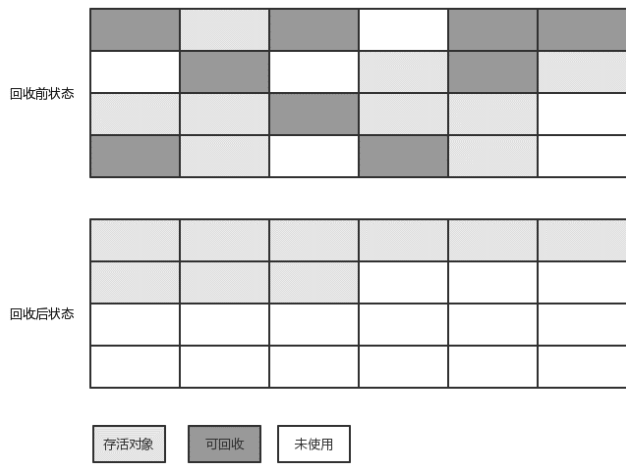


图 10-10 标记—压缩算法的执行过程

标记—压缩算法解决了标记—清除算法效率低和容易产生大量内存碎片的问题，它被广泛应用于老年代中。

10.8.4 分代收集算法

分代收集算法会结合不同的收集算法来处理不同的空间，因此在学习分代收集算法之前我们首先要了解 Java 堆区的空间划分。Java 堆区的空间划分在 Java 虚拟机中，各种对象的生命周期会有着较大的差别，大部分对象生命周期很短暂，少部分对象生命周期很长，有的甚至与应用程序以及 Java 虚拟机的运行周期一样长。因此，应该对不同生命周期的对象采取不同的收集策略，根据生命周期长短将它们分别放到不同的区域，并在不同的区域采用不同的收集算法，这就是分代的概念。现在主流的 Java 虚拟机的垃圾收集器都采用分代收集算法（Generational Collection）。Java 堆区基于分代的概念，分为新生代（Young Generation）和老年代（Tenured Generation），其中新生代再细分为 Eden 空间、From Survivor 空间和 To Survivor 空间。因为 Eden 空间中的大多数对象生命周期很短，所以新生代的空间划分并不是均分的，HotSpot 虚拟机默认 Eden 空间和两个 Survivor 空间的所占的比例为 8:1。

1. 分代收集

根据 Java 堆区的空间划分，垃圾收集的类型分为两种，它们分别如下。

- Minor Collection：新生代垃圾收集。
- Full Collection：对老年代进行收集，又可以称作 Major Collection，Full Collection 通常情况下会伴随至少一次的 Minor Collection，它的收集频率较低，耗时较长。

当执行一次 Minor Collection 时，Eden 空间的存活对象会被复制到 To Survivor 空间，并且之前经过一次 Minor Collection 并在 From Survivor 空间存活的仍年轻的对象也会复制到 To Survivor 空间。有两种情况 Eden 空间和 From Survivor 空间存活的对象不会复制到 To Survivor 空间，而是晋升到老年代。一种是存活的对象的分代年龄超过 `-XX:MaxTenuringThreshold`（用于控制对象经历多少次 Minor GC 才晋升到老年代）所指定的阈值。另一种是 To Survivor 空间容量达到阈值。当所有存活的对象被复制到 To Survivor 空间，或者晋升到老年代，也就意味着 Eden 空间和 From Survivor 空间剩下的都是可回收对象，如图 10-11 所示。

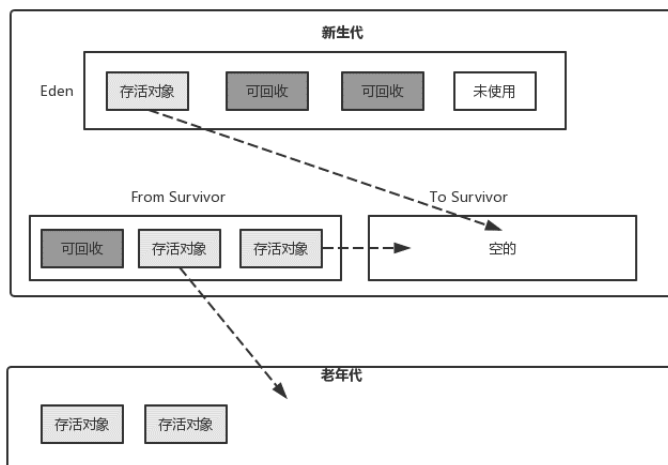


图 10-11 复制算法在新生代中的应用

这个时候 GC 执行 Minor Collection，Eden 空间和 From Survivor 空间都会被清空，新生代中存活的对象都存放在 To Survivor 空间。接下来将 From Survivor 空间和 To Survivor 空间互换位置，也就是此前的 From Survivor 空间成为了现在的 To Survivor 空间，每次 Survivor 空间互换都要保证 To Survivor 空间是空的，这就是复制算法在新生代中的应用。在老年代则会采用标记—压缩算法或者标记—清除算法。

10.9 本章小结

Java 虚拟机是一个很庞大的知识体系，本章也只是介绍了 Java 虚拟机知识中的非常少的一部分，包括 Java 虚拟机结构、oop-klass 模型、垃圾标记算法和垃圾收集算法等。对于 Android 开发者来说，了解这些知识点对普通的 Android 开发工作完全够用，如果想要更深入地了解 Java 虚拟机则需要阅读专业介绍 Java 虚拟机的书籍。

第 11 章

Dalvik 和 ART

关联章节：第 10 章 Java 虚拟机

第 10 章我们学习了 Java 虚拟机，这一章我们顺势来学习 Dalvik 和 ART，关于这两个虚拟机的知识体系也是十分庞大的，对于 Android 应用开发来说，掌握它们的基本原理并且会看它们的 Log 就可以了，因此本章只介绍 Dalvik 和 ART 的基础。

11.1 Dalvik虚拟机

Dalvik 虚拟机 (Dalvik Virtual Machine)，简称 Dalvik VM 或者 DVM。它是由 Dan Bornstein 编写的，名字源于他的祖先居住过的名为 Dalvik 的小渔村。DVM 是 Google 专门为 Android 平台开发的虚拟机，它运行在 Android 运行时库中。需要注意的是 DVM 并不是一个 Java 虚拟机（以下简称 JVM），至于为什么，下文会给你答案。

11.1.1 DVM与JVM的区别

DVM 之所以不是一个 JVM，主要原因是 DVM 并没有遵循 JVM 规范来实现，DVM 与 JVM 主要有以下区别。

1. 基于的架构不同

JVM 基于栈则意味着需要去栈中读写数据，所需的指令会更多，这样会导致速度变慢，

对于性能有限的移动设备，显然不是很适合的。DVM 是基于寄存器的，它没有基于栈的虚拟机在复制数据时而使用的大量的出入栈指令，同时指令更紧凑、更简洁。但是由于显式指定了操作数，所以基于寄存器的指令会比基于栈的指令要大，但是由于指令数量的减少，总的代码数不会增加多少。

2. 执行的字节码不同

在 Java SE 程序中，Java 类被编译成一个或多个.class 文件，并打包成 jar 文件，而后 JVM 会通过相应的.class 文件和 jar 文件获取相应的字节码。执行顺序为.java 文件→.class 文件→.jar 文件，而 DVM 会用 dx 工具将所有的.class 文件转换为一个.dex 文件，然后 DVM 会从该.dex 文件读取指令和数据。执行顺序为.java 文件→.class 文件→.dex 文件。

如图 11-1 所示，.jar 文件里面包含多个.class 文件，每个.class 文件里面包含了该类的常量池、类信息、属性等。当 JVM 加载该.jar 文件的时候，会加载里面的所有的.class 文件，JVM 的这种加载方式很慢，对于内存有限的移动设备并不合适。而在.apk 文件中只包含了一个.dex 文件，这个.dex 文件将所有的.class 里面所包含的信息全部整合在一起了，这样再加载就加快了速度。.class 文件存在很多的冗余信息，dex 工具会去除冗余信息，并把所有的.class 文件整合到.dex 文件中，减少了 I/O 操作，加快了类的查找速度。

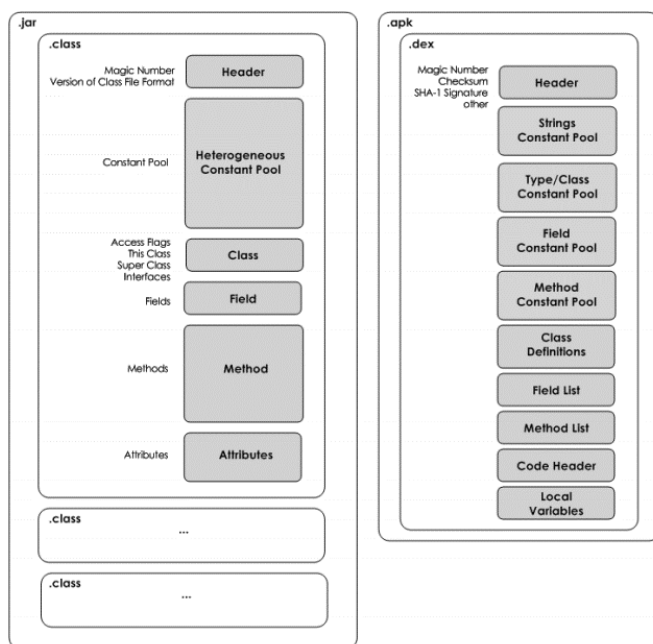


图 11-1 执行的字节码不同

3. DVM 允许在有限的内存中同时运行多个进程

DVM 经过优化，允许在有限的内存中同时运行多个进程。在 Android 中的每一个应用都运行在一个 DVM 实例中，每一个 DVM 实例都运行在一个独立的进程空间中，独立的进程可以防止在虚拟机崩溃的时候所有程序都被关闭。

4. DVM 由 Zygote 创建和初始化

我们在第 2 章学习过 Zygote，它是一个 DVM 进程，同时也用来创建和初始化 DVM 实例。每当系统需要创建一个应用程序时，Zygote 就会 fork 自身，快速地创建和初始化一个 DVM 实例，用于应用程序的运行。对于一些只读的系统库，所有的 DVM 实例都会和 Zygote 共享一块内存区域，节省了内存开销。

5. DVM 有共享机制

DVM 拥有预加载—共享的机制，不同应用之间在运行时可以共享相同的类，拥有更高的效率。而 JVM 机制不存在这种共享机制，不同的程序，打包以后的程序都是彼此独立的，即便它们在包里使用了同样的类，运行时也都是单独加载和运行的，无法进行共享。

6. DVM 早期没有使用 JIT 编译器

JVM 使用了 JIT 编译器（Just In Time Compiler，即时编译器），而 DVM 早期没有使用 JIT 编译器。早期的 DVM 每次执行代码，都需要通过解释器将 dex 代码编译成机器码，然后交给系统处理，效率不是很高。为了解决这一问题，从 Android 2.2 版本开始 DVM 使用了 JIT 编译器，它会对多次运行的代码（热点代码）进行编译，生成相当精简的本地机器码（Native Code），这样在下次执行到相同逻辑的时候，直接使用编译之后的本地机器码，而不是每次都需要编译。需要注意的是，应用程序每一次重新运行的时候，都要重做这个编译工作，因此每次重新打开应用程序，都需要 JIT 编译。

11.1.2 DVM架构

DVM 的源码位于 dalvik/目录下，Android 8.0 中的 DVM 源码的部分目录说明如表 11-1 所示。

表 11-1 DVM 的源码目录

目录/文件	说 明
dexdump	生成 dex 文件的反编译查看工具，主要用来查看编译出来的代码的正确性和结构
dexgen	dex 代码生成器项目

续表

目录/文件	说 明
docs	DVM 相关帮助文档
dx	Java 字节码转换为 DVM 机器码的工具
libdex	生成主机和设备处理 dex 文件的库
tools	一些编译和运行相关的工具
Android.mk	虚拟机编译的 makefile 配置文件
MODULE_LICENSE_APACHE2	APACHE2 版权声明文件
NOTICE	虚拟机源码版权注意事项文件

其中，dalvik/libdex 会被编译成 libdex.a 静态库，作为 dex 工具使用；dalvik/dexdump 是 .dex 文件的反编译工具，DVM 架构如图 11-2 所示。

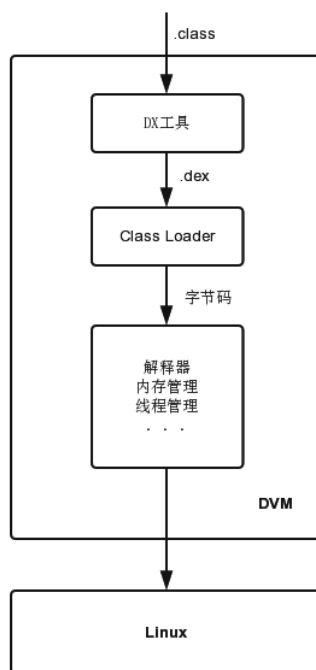


图 11-2 DVM 架构

从图 11-2 可以看出，首先 Java 编译器编译的 .class 文件经过 DX 工具转换为 .dex 文件，.dex 文件由类加载器处理，接着解释器根据指令集对 Dalvik 字节码进行解释、执行，最后交于 Linux 处理。

11.1.3 DVM的运行时堆

DVM 的运行时堆使用标记—清除 (Mark-Sweep) 算法进行 GC，它由两个 Space 以及多个辅助数据结构组成，两个 Space 分别是 Zygote Space (Zygote Heap) 和 Allocation Space (Active Heap)。Zygote Space 用来管理 Zygote 进程在启动过程中预加载和创建的各种对象，Zygote Space 中不会触发 GC，在 Zygote 进程和应用程序进程之间会共享 Zygote Space。在 Zygote 进程 fork 第一个子进程之前，会把 Zygote Space 分为两个部分，原来的已经被使用的那部分堆仍旧叫 Zygote Space，而未使用的那部分堆就叫 Allocation Space，以后的对象都会在 Allocation Space 上进行分配和释放。Allocation Space 不是进程间共享的，在每个进程中都独立拥有一份。除了这两个 Space，还包含以下数据结构。

- Card Table：用于 DVM Concurrent GC，当第一次进行垃圾标记后，记录垃圾信息。
- Heap Bitmap：有两个 Heap Bitmap，一个用来记录上次 GC 存活的对象，另一个用来记录这次 GC 存活的对象。
- Mark Stack：DVM 的运行时堆使用标记—清除 (Mark-Sweep) 算法进行 GC，Mark Stack 就是在 GC 的标记阶段使用的，它用来遍历存活的对象。

11.1.4 DVM的GC日志

在 10.6.2 节中提到了 Java 虚拟机的 GC 日志。DVM 和 ART 的 GC 日志与 Java 虚拟机的日志有较大的区别。在 DVM 中每次垃圾收集都会将 GC 日志打印到 logcat 中，具体的格式为：

```
D/dalvikvm: <GC_Reason> <Amount_freed>, <Heap_stats>, <External_memory_stats>,
<Pause_time>
```

可以看到 DVM 的日志共有 5 个信息，其中 GC Reason 有很多种，这里将它单独拿出来进行介绍。

1. 引起 GC 的原因

GC Reason 就是引起 GC 的原因，有以下几种。

- GC_CONCURRENT：当堆开始填充时，并发 GC 可以释放内存。
- GC_FOR_MALLOC：当堆内存已满时，App 尝试分配内存而引起的 GC，系统必须停止 App 并回收内存。
- GC_HPROF_DUMP_HEAP：当你请求创建 HPROF 文件来分析堆内存时出现的 GC。

- GC_EXPLICIT: 显式的 GC, 例如调用 `System.gc()` (应该避免调用显式的 GC, 信任 GC 会在需要时运行)。
- GC_EXTERNAL_ALLOC: 仅适用于 API 级别小于等于 10, 且用于外部分配内存的 GC。

2. 其他的信息

除了引起 GC 的原因, 其他的信息如下。

- Amount_freed: 本次 GC 释放内存的大小。
- Heap_stats: 堆的空闲内存百分比 (已用内存) / (堆的总内存)。
- External_memory_stats: API 小于等于级别 10 的内存分配 (已分配的内存) / (引起 GC 的阈值)。
- Pause time: 暂停时间, 更大的堆会有更长的暂停时间。并发暂停时间会显示两个暂停时间, 即一个出现在垃圾收集开始时, 另一个出现在垃圾收集快要完成时。

3. 实例分析

为了让大家更好地理解 DVM 的 GC 日志, 举一个具体的 GC 日志实例, 如下所示:

```
D/dalvikvm: GC_CONCURRENT freed 2012K, 63% free 3213K/9291K, external 4501K/5161K,
paused 2ms+2ms
```

这个 GC 日志的含义为: 引起 GC 的原因是 GC_CONCURRENT; 本次 GC 释放的内存为 2012KB; 堆的空闲内存百分比为 63%, 已用内存为 3213KB, 堆的总内存为 9291KB; 暂停的总时长为 4ms。

11.2 ART虚拟机

ART (Android Runtime) 虚拟机是 Android 4.4 发布的, 用来替换 Dalvik 虚拟机, Android 4.4 默认采用的还是 DVM, 系统会提供一个选项来开启 ART。在 Android 5.0 版本中默认采用了 ART, DVM 从此退出历史舞台。

11.2.1 ART与DVM的区别

ART 和 DVM 的区别主要有如下 4 点。

(1) 从 11.1 节我们知道, DVM 中的应用每次运行时, 字节码都需要通过 JIT 编译器编译为机器码, 这会使得应用程序的运行效率降低。而在 ART 中, 系统在安装应用程序时会进行一次 AOT (ahead of time compilation, 预编译), 将字节码预先编译成机器码并存储在本地, 这样应用程序每次运行时就不需要执行编译了, 运行效率会大大提升, 设备的耗电量也会降低。这就好比我们在线阅读漫画, DVM 是我们阅读到哪就加载哪, ART 则是直接加载一章的漫画, 虽然一开始加载速度有些慢, 但是后续的阅读体验会很流畅。采用 AOT 也会有缺点, 主要有两个: 第一个是 AOT 会使得应用程序的安装时间变长, 尤其是一些复杂的应用; 第二个是字节码预先编译成机器码, 机器码需要的存储空间会多一些。为了解决上面的缺点, Android 7.0 版本中的 ART 加入了即时编译器 JIT, 作为 AOT 的一个补充, 在应用程序安装时并不会将字节码全部编译成机器码, 而是在运行中将热点代码编译成机器码, 从而缩短应用程序的安装时间并节省了存储空间。

(2) DVM 是为 32 位 CPU 设计的, 而 ART 支持 64 位并兼容 32 位 CPU, 这也是 DVM 被淘汰的主要原因之一。

(3) ART 对垃圾回收机制进行了改进, 比如更频繁地执行并行垃圾收集, 将 GC 暂停由 2 次减少为 1 次等。

(4) ART 的运行时堆空间划分和 DVM 不同。

11.2.2 ART 的运行时堆

与 DVM 的 GC 不同的是, ART 采用了多种垃圾收集方案, 每个方案会运行不同的垃圾收集器, 默认是采用了 CMS (Concurrent Mark-Sweep) 方案, 该方案主要使用了 sticky-CMS 和 partial-CMS。根据不同的 CMS 方案, ART 的运行时堆的空间也会有不同的划分, 默认是由 4 个 Space 和多个辅助数据结构组成的, 4 个 Space 分别是 Zygote Space、Allocation Space、Image Space 和 Large Object Space。Zygote Space、Allocation Space 和 DVM 中的作用是一样的, Image Space 用来存放一些预加载类, Large Object Space 用来分配一些大对象 (默认大小为 12KB), 其中 Zygote Space 和 Image Space 是进程间共享的。采用标记—清除算法的运行时堆空间划分如图 11-3 所示。

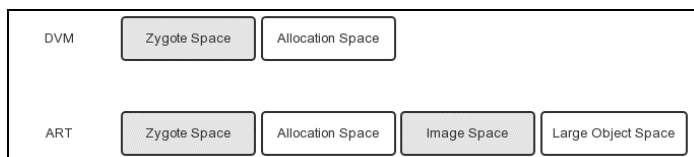


图 11-3 采用标记—清除算法的运行时堆空间划分

除了这四个 Space，ART 的 Java 堆中还包括两个 Mod Union Table，一个 Card Table，两个 Heap Bitmap，两个 Object Map，以及三个 Object Stack。

11.2.3 ART 的 GC 日志

ART 的 GC 日志与 DVM 不同，ART 会为那些主动请求的垃圾收集事件或者认为 GC 速度慢时才会打印 GC 日志。GC 速度慢指的是 GC 暂停超过 5ms 或者 GC 持续时间超过 100ms。如果 App 未处于可察觉的暂停进程状态，那么它的 GC 不会被认为是慢速的。

ART 的 GC 日志具体的格式为：

```
I/art: <GC_Reason> <GC_Name> <Objects_freed>(<Size_freed>) AllocSpace Objects,
      <Large_objects_freed>(<Large_object_size_freed>) <Heap_stats> LOS objects,
      <Pause_time(s)>
```

下面对 GC 日志的组成部分进行介绍。

1. 引起 GC 原因

ART 的引起 GC 原因 (GC_Reason) 要比 DVM 多一些，有以下几种。

- Concurrent：并发 GC，不会使 App 的线程暂停，该 GC 是在后台线程运行的，并不会阻止内存分配。
- Alloc：当堆内存已满时，App 尝试分配内存而引起的 GC，这个 GC 会发生在正在分配内存的线程中。
- Explicit：App 显示的请求垃圾收集，例如调用 System.gc()。与 DVM 一样，最佳做法是应该信任 GC 并避免显式地请求 GC，显式地请求 GC 会阻止分配线程并不必要地浪费 CPU 周期。如果显式地请求 GC 导致其他线程被抢占，那么有可能导致 jank (App 同一帧画了多次)。
- NativeAlloc：Native 内存分配时，比如为 Bitmaps 或者 RenderScript 分配对象，这会导致 Native 内存压力，从而触发 GC。
- CollectorTransition：由堆转换引起的回收，这是运行时切换 GC 而引起的。收集器转换包括将所有对象从空闲列表空间复制到碰撞指针空间（反之亦然）。当前，收集器转换仅在以下情况下出现：在内存较小的设备上，App 将进程状态从可察觉的暂停状态变更为可察觉的非暂停状态（反之亦然）。
- HomogeneousSpaceCompact：齐性空间压缩是指空闲列表到压缩的空闲列表空间，通常发生在当 App 已经移动到可察觉的暂停进程状态时。这样做的主要原因是减少

了内存使用并对堆内存进行碎片整理。

- DisableMovingGc：不是真正触发 GC 的原因，发生并发堆压缩时，由于使用了 GetPrimitiveArrayCritical，收集会被阻塞。在一般情况下，强烈建议不要使用 GetPrimitiveArrayCritical，因为它在移动收集器方面具有限制。
- HeapTrim：不是触发 GC 的原因，但是请注意，收集会一直被阻塞，直到堆内存整理完毕。

2. 垃圾收集器名称

GC_Name 指的是垃圾收集器名称，有以下几种。

- Concurrent Mark Sweep (CMS)：CMS 收集器是一种以获取最短收集暂停时间为目标的收集器，采用了标记—清除算法实现。它是完整的堆垃圾收集器，能释放除了 Image Space 外的所有的空间。
- Concurrent Partial Mark Sweep：部分完整的堆垃圾收集器，能释放除了 Image Space 和 Zygote Space 外的所有空间。
- Concurrent Sticky Mark Sweep：粘性收集器，基于分代的垃圾收集思想，它只能释放自上次 GC 以来分配的对象。这个垃圾收集器比一个完整的或部分完整的垃圾收集器扫描得更频繁，因为它更快并且有更短的暂停时间。
- Marksweep + Semispace：非并发的 GC，复制 GC 用于堆转换以及齐性空间压缩（堆碎片整理）。

3. 其他信息

- Objects freed：本次 GC 从非 Large Object Space 中回收的对象的数量。
- Size_freed：本次 GC 从非 Large Object Space 中回收的字节数。
- Large objects freed：本次 GC 从 Large Object Space 中回收的对象的数量。
- Large object size freed：本次 GC 从 Large Object Space 中回收的字节数。
- Heap stats：堆的空闲内存百分比，即（已用内存）/（堆的总内存）。
- Pause times：暂停时间，暂停时间与在 GC 运行时修改的对象引用的数量成比例。目前，ART 的 CMS 收集器仅有一次暂停，它出现在 GC 的结尾附近。移动的垃圾收集器暂停时间会很长，会在大部分垃圾回收期间持续出现。

4. 实例分析

```
I/art : Explicit concurrent mark sweep GC freed 104710(7MB) AllocSpace objects,
21(416KB) LOS objects, 33% free, 25MB/38MB, paused 1.230ms total 67.216ms
```

这个 GC 日志的含义为引起 GC 原因是 Explicit；垃圾收集器为 CMS 收集器；释放对象的数量为 104710 个，释放字节数为 7MB；释放大对象的数量为 21 个，释放大对象字节数为 416KB；堆的空闲内存百分比为 33%，已用内存为 25MB，堆的总内存为 38MB；GC 暂停时长为 1.230ms，GC 总时长为 67.216ms。

11.3 DVM和ART的诞生

虽然 DVM 和 ART 的知识体系非常庞大,但是我们仍旧有必要了解 DVM 是怎么来的。在 2.1.5 节中讲过 init 启动 Zygote 时会调用 app_main.cpp 的 main 函数，如下所示：

frameworks/base/cmds/app_process/app_main.cpp

```
int main(int argc, char* const argv[])
{
    ...
    if (zygote) { //1
        runtime.start("com.android.internal.os.ZygoteInit", args, zygote); //2
    } else if (className) {
        runtime.start("com.android.internal.os.RuntimeInit", args, zygote);
    } else {
        fprintf(stderr, "Error: no class name or --zygote supplied.\n");
        app_usage();
        LOG_ALWAYS_FATAL("app_process: no class name or --zygote supplied.");
        return 10;
    }
}
```

在注释 1 处如果为 true，就说明当前程序运行在 Zygote 进程中，在注释 2 处调用 AppRuntime 的 start 函数，start 函数具体在 AppRuntime 的父类 AndroidRuntime 中实现，如下所示：

frameworks/base/core/jni/AndroidRuntime.cpp

```
void AndroidRuntime::start(const char* className, const Vector<String8>& options,
bool zygote)
{
    ...
    JniInvocation jni_invocation;
    jni_invocation.Init(NULL); //1
    JNIEnv* env;
    //启动 Java 虚拟机
```

```

    if (startVm(&mJavaVM, &env, zygote) != 0) { //2
        return;
    }
    onVmCreated(env);
    //为 Java 虚拟机注册 JNI 方法
    if (startReg(env) < 0) { //3
        ALOGE("Unable to register all android natives\n");
        return;
    }
    ...
}

```

在注释 2 处调用 startVm 函数来创建 Java 虚拟机，在注释 3 处调用 startReg 函数来为 Java 虚拟机注册 JNI 方法。在注释 1 处调用了 jni_invocation 的 Init 函数：

libnativehelper/JniInvocation.cpp

```

bool JniInvocation::Init(const char* library) {
#ifdef __ANDROID__
    char buffer[PROP_VALUE_MAX];
#else
    char* buffer = NULL;
#endif
    library = GetLibrary(library, buffer); //1
    const int kDlopenFlags = RTLD_NOW | RTLD_NODELETE;
    handle_ = dlopen(library, kDlopenFlags); //2
    if (handle_ == NULL) {
        if (strcmp(library, kLibraryFallback) == 0) {
            ALOGE("Failed to dlopen %s: %s", library, dlerror());
            return false;
        }
        ...
    }
    return true;
}

```

在注释 1 处调用了 GetLibrary 函数：

libnativehelper/JniInvocation.cpp

```

#ifdef __ANDROID__ //1
static const char* kLibrarySystemProperty = "persist.sys.dalvik.vm.lib.2"; //2
static const char* kDebuggableSystemProperty = "ro.debuggable";
#endif
static const char* kLibraryFallback = "libart.so";

template<typename T> void UNUSED(const T&) {}

```

```

const char* JNIInvocation::GetLibrary(const char* library, char* buffer) {
#ifdef __ANDROID__
    const char* default_library;
    char debuggable[PROP_VALUE_MAX];
    __system_property_get(kDebuggableSystemProperty, debuggable);
    if (strcmp(debuggable, "1") != 0) { //3
        library = kLibraryFallback; //4
        default_library = kLibraryFallback; //5
    } else {
        if (buffer != NULL) {
            if (__system_property_get(kLibrarySystemProperty, buffer) > 0) { //6
                default_library = buffer;
            } else {
                default_library = kLibraryFallback;
            }
        } else {
            default_library = kLibraryFallback;
        }
    }
}
#else
    UNUSED(buffer);
    const char* default_library = kLibraryFallback; //7
#endif
    if (library == NULL) {
        library = default_library; //8
    }

    return library;
}

```

注释 1 处代表在 Android 平台，注释 2 处的 `persist.sys.dalvik.vm.lib.2` 是一个系统属性，它的取值可以为 `libdvm.so` 或者 `libart.so`，值为 `libdvm.so` 说明当前用的是 DVM，值为 `libart.so` 说明当前用的是 ART。在注释 3 处如果 `debuggable` 不等于“1”，说明当前不是 Debug 模式构建的，是不允许动态更改虚拟机动态库的。在注释 4 和注释 5 处将 `libart.so` 赋值给 `library` 和 `default_library`。如果是 Debug 模式构建会在注释 6 处读取 `persist.sys.dalvik.vm.lib.2` 配置中是否有传入的参数 `buffer`，如果有就将 `default_library` 赋值为 `buffer`，如果没有将 `default_library` 赋值为 `libart.so`。在注释 7 处如果不是在 Android 平台，`default_library` 的值为 `libart.so`。在注释 8 处如果 `library` 为 `NULL` 就将 `default_library` 赋值给 `library` 并返回该 `library`。这里我们知道 Android 8.0 如果不是 Debug 模式构建，只能返回 `libart.so`。回到 `JNIInvocation` 的 `Init` 函数，注释 1 处的 `GetLibrary` 函数会返回 `libart.so` 或者 `libdvm.so`，接着在注释 2 处调用 `dlopen` 函数来加载 `libart.so` 或者 `libdvm.so`。因此我们知道 `JNIInvocation` 的

Init 函数的主要作用是初始化 ART 或者 DVM 的环境,初始化完毕后会调用 startVm 函数来启动相应的虚拟机。讲到这里我们应该知道 DVM 和 ART 是如何诞生的了,没错,是在 Zygote 进程中诞生的,这样 Zygote 进程就持有了 DVM 或者 ART 的实例,此后 Zygote 进程 fork 自身创建应用程序进程时,应用程序进程也得到了 DVM 或者 ART 的实例,这样就不需要每次启动应用程序进程都要创建 DVM 或者 ART,从而加快了应用程序进程的启动速度。

11.4 本章小结

本章介绍了 DVM 和 ART 的基本原理、如何阅读它们的 log 和 DVM,以及 ART 的诞生。阅读本章前请阅读第 2 章、第 3 章和第 10 章会有利于对本章的理解。DVM 和 ART 的知识体系完全可以写一本书,如果想要更多地了解它们请阅读专业的书籍和博客,博客推荐老罗(罗升阳)的博客,里面有一系列文章专门介绍 DVM 和 ART,虽然文章基于的 Android 版本有些老,但仍具有参考价值。

第 12 章

理解 ClassLoader

关联章节：第 2 章 Android 系统启动；第 10 章 Java 虚拟机

热修复和插件化是目前比较热门的技术,要想更好地掌握它们需要先了解 ClassLoader。关于 ClassLoader,可能有的同学会认为 Java 中的 ClassLoader 和 Android 中的 ClassLoader 没有区别,在第 11 章中我们知道 DVM 和 ART 加载的是 dex 文件,而 JVM 加载的是 Class 文件,因此它们的类加载器 ClassLoader 肯定是有区别的。这一章分别介绍 Java 中的 ClassLoader 和 Android 中的 ClassLoader,这样它们的区别也就一目了然了。

12.1 Java中的ClassLoader

在 10.2.3 节中提到过类加载子系统,它的主要作用就是通过多种类加载器 (ClassLoader) 来查找和加载 Class 文件到 Java 虚拟机中。

12.1.1 ClassLoader的类型

Java 中的类加载器主要有两种类型,即系统类加载器和自定义类加载器。其中系统类加载器包括 3 种,分别是 Bootstrap ClassLoader、Extensions ClassLoader 和 Application ClassLoader。

1. Bootstrap ClassLoader（引导类加载器）

C/C++代码实现的加载器，用于加载指定的 JDK 的核心类库，比如 `java.lang.`、`java.util.` 等这些系统类。它用来加载以下目录中的类库：

- `$JAVA_HOME/jre/lib` 目录。
- `-Xbootclasspath` 参数指定的目录。

Java 虚拟机的启动就是通过 Bootstrap ClassLoader 创建一个初始类来完成的。由于 Bootstrap ClassLoader 是使用 C/C++语言实现的，所以该加载器不能被 Java 代码访问到。需要注意的是，Bootstrap ClassLoader 并不继承 `java.lang.ClassLoader`。我们可以通过如下代码来得出 Bootstrap ClassLoader 所加载的目录：

```
public class ClassLoaderTest {  
    public static void main(String[] args) {  
        System.out.println(System.getProperty("sun.boot.class.path"));  
    }  
}
```

打印结果为：

```
C:\Program Files\Java\jdk1.8.0_102\jre\lib\resources.jar;C:\Program Files\Java\jdk1.8.0_102\jre\lib\rt.jar; C:\Program Files\Java\jdk1.8.0_102\jre\lib\sunrsasign.jar; C:\Program Files\Java\jdk1.8.0_102\jre\lib\jsse.jar;C:\Program Files\Java\jdk1.8.0_102\jre\lib\jce.jar; C:\Program Files\Java\jdk1.8.0_102\jre\lib\charsets.jar; C:\Program Files\Java\jdk1.8.0_102\jre\lib\jfr.jar;C:\Program Files\Java\jdk1.8.0_102\jre\classes
```

可以发现几乎都是 `$JAVA_HOME/jre/lib` 目录中的 jar 包，包括 `rt.jar`、`resources.jar` 和 `charsets.jar` 等。

2. Extensions ClassLoader（拓展类加载器）

Java 中的实现类为 `ExtClassLoader`，因此可以简称为 `ExtClassLoader`，它用于加载 Java 的拓展类，提供除了系统类之外的额外功能。`ExtClassLoader` 用来加载以下目录中的类库：

- 加载 `$JAVA_HOME/jre/lib/ext` 目录。
- 系统属性 `java.ext.dir` 所指定的目录。

通过以下代码可以得到 Extensions ClassLoader 加载目录：

```
System.out.println(System.getProperty("java.ext.dirs"));
```

打印结果为：


```
C:\Program Files\Java\jdk1.8.0_102\jre\lib\ext; C:\Windows\Sun\Java\lib\ext
```

3. Application ClassLoader（应用程序类加载器）

Java 中的实现类为 `AppClassLoader`，因此可以简称为 `AppClassLoader`，同时它又可以称作 `System ClassLoader`（系统类加载器），这是因为 `AppClassLoader` 可以通过 `ClassLoader` 的 `getSystemClassLoader` 方法获取到。它用来加载以下目录中的类库：

- 当前程序的 Classpath 目录。
- 系统属性 `java.class.path` 指定的目录。

4. Custom ClassLoader（自定义类加载器）

除了系统提供的类加载器，还可以自定义类加载器，自定义类加载器通过继承 `java.lang.ClassLoader` 类的方式来实现自己的类加载器，`Extensions ClassLoader` 和 `App ClassLoader` 也继承了 `java.lang.ClassLoader` 类。关于自定义类加载器后面会进行介绍。

12.1.2 ClassLoader的继承关系

运行一个 Java 程序需要用到几种类型的类加载器呢？如下所示。

```
public class ClassLoaderTest {
    public static void main(String[] args) {
        ClassLoader loader = ClassLoaderTest.class.getClassLoader();
        while (loader != null) {
            System.out.println(loader); //1
            loader = loader.getParent();
        }
    }
}
```

我们得到当前类 `ClassLoaderTest` 的类加载器，并在注释 1 处打印出来，紧接着打印出当前类的类加载器的父加载器，直到没有父加载器时就终止循环，打印结果如下所示：

```
sun.misc.Launcher$AppClassLoader@75b84c92
sun.misc.Launcher$ExtClassLoader@1b6d3586
```

第 1 行说明加载 `ClassLoaderTest` 的类加载器是 `AppClassLoader`，第 2 行说明 `AppClassLoader` 的父加载器为 `ExtClassLoader`。至于为何没有打印出 `ExtClassLoader` 的父加载器 `Bootstrap ClassLoader`，这是因为 `Bootstrap ClassLoader` 是由 C/C++ 编写的，并不是一个 Java 类，因此我们无法在 Java 代码中获取它的引用。系统所提供的类加载器有 3 种类型，但是系统提供的 `ClassLoader` 却不只有 3 个。另外，`AppClassLoader` 的父类加载器为

ExtClassLoader, 并不代表 AppClassLoader 继承自 ExtClassLoader, ClassLoader 的继承关系如图 12.1 所示。

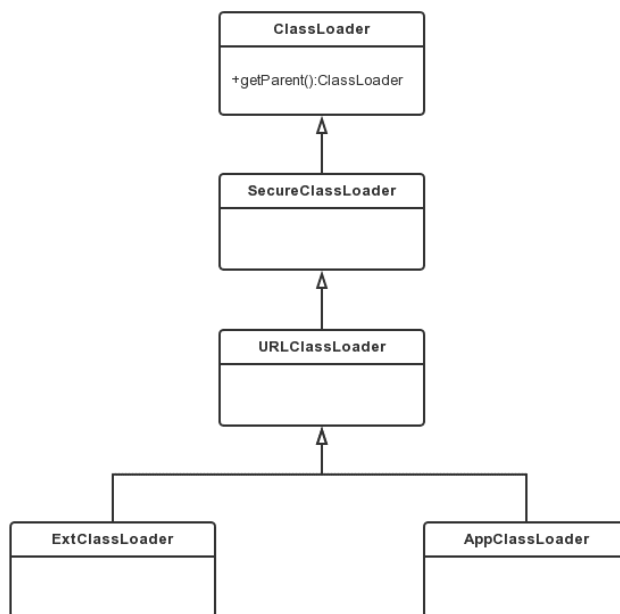


图 12-1 ClassLoader 的继承关系

可以看到图 12-1 中共有 5 个 ClassLoader 相关类, 下面简单对它们进行介绍。

- ClassLoader 是一个抽象类, 其中定义了 ClassLoader 的主要功能。
- SecureClassLoader 继承了抽象类 ClassLoader, 但 SecureClassLoader 并不是 ClassLoader 的实现类, 而是拓展了 ClassLoader 类加入了权限方面的功能, 加强了 ClassLoader 的安全性。
- URLClassLoader 继承自 SecureClassLoader, 可以通过 URL 路径从 jar 文件和文件夹中加载类和资源。
- ExtClassLoader 和 AppClassLoader 都继承自 URLClassLoader, 它们都是 Launcher 的内部类, Launcher 是 Java 虚拟机的入口应用, ExtClassLoader 和 AppClassLoader 都是在 Launcher 中进行初始化的。

12.1.3 双亲委托模式

类加载器查找 Class 所采用的是双亲委托模式, 所谓双亲委托模式就是首先判断该

Class 是否已经加载, 如果没有则不是自身去查找而是委托给父加载器进行查找, 这样依次进行递归, 直到委托到最顶层的 Bootstrap ClassLoader, 如果 Bootstrap ClassLoader 找到了该 Class, 就会直接返回, 如果没找到, 则继续依次向下查找, 如果还没找到则最后会交由自身去查找。这样讲可能会有些抽象, 来看图 12-2。

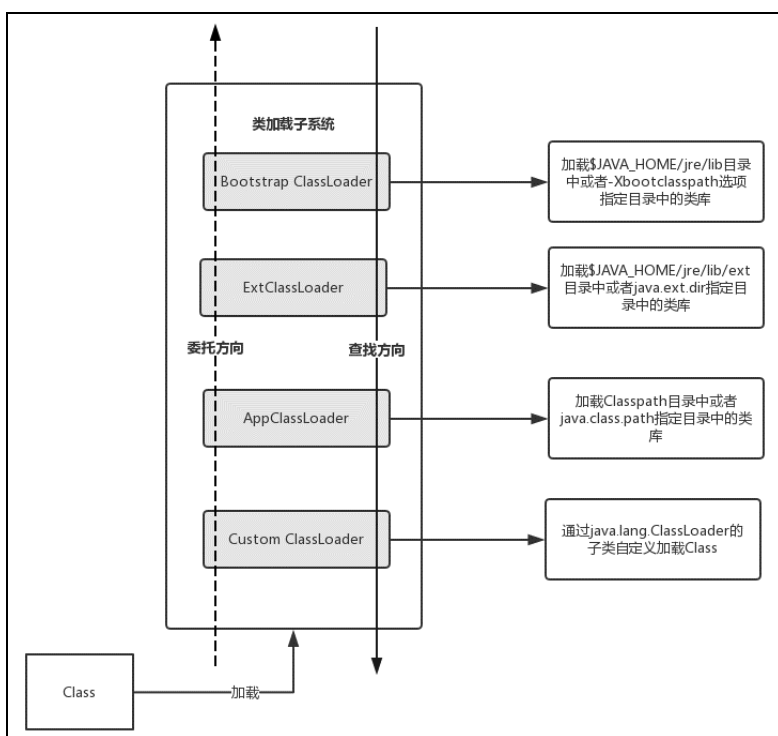


图 12-2 双亲委托模式

我们知道类加载子系统用来查找和加载 Class 文件到 Java 虚拟机中, 假设我们要加载一个位于 D 盘的 Class 文件, 这时系统所提供的类加载器不能满足条件, 这时就需要我们自定义类加载器继承自 `java.lang.ClassLoader`, 并复写它的 `findClass` 方法。加载 D 盘的 Class 文件步骤如下:

(1) 自定义类加载器首先从缓存中查找 Class 文件是否已经加载, 如果已经加载就返回该 Class, 如果没加载则委托给父加载器也就是 AppClassLoader。

(2) 按照图 12-2 中虚线的方向递归步骤 1。

(3) 一直委托到 Bootstrap ClassLoader, 如果 Bootstrap ClassLoader 查找缓存也没有加载 Class 文件, 则在 `$JAVA_HOME/jre/lib` 目录中或者 `--Xbootclasspath` 参数指定的目录中进

行查找，如果找到就加载并返回该 Class，如果没有找到则交给子加载器 ExtClassLoader。

(4) ExtClassLoader 在 \$JAVA_HOME/jre/lib/ext 目录中或者系统属性 java.ext.dir 所指定的目录中进行查找，如果找到就加载并返回，找不到则交给 AppClassLoader。

(5) AppClassLoade 在 Classpath 目录中或者系统属性 java.class.path 指定的目录中进行查找，如果找到就加载并返回，找不到交给我们自定义的类加载器，如果还找不到则抛出异常。

总的来说就是 Class 文件加载到类加载子系统后，先沿着图 12-2 中虚线的方向自下而上进行委托，再沿着实线的方向自上而下进行查找和加载，整个过程就是先上后下。结合 12.1.2 节中讲的 ClassLoader 的继承关系，可以得出 ClassLoader 的父子关系并不是使用继承来实现的，而是使用组合来实现代码复用的。

类加载的步骤在 JDK8 的源码中也得到了体现，下面来查看抽象类 ClassLoader 的 loadClass 方法：

```
protected Class<?> More ...loadClass(String name, boolean resolve)
throws ClassNotFoundException
{
    synchronized (getClassLoadingLock(name)) {
        Class<?> c = findLoadedClass(name);//1
        if (c == null) {
            long t0 = System.nanoTime();
            try {
                if (parent != null) {
                    c = parent.loadClass(name, false);//2
                } else {
                    c = findBootstrapClassOrNull(name);//3
                }
            } catch (ClassNotFoundException e) {
            }
            if (c == null) {
                long t1 = System.nanoTime();
                c = findClass(name);//4
                sun.misc.PerfCounter.getParentDelegationTime().addTime(t1 - t0);
                sun.misc.PerfCounter.getFindClassTime().addElapsedTimeFrom(t1);
                sun.misc.PerfCounter.getFindClasses().increment();
            }
        }
        if (resolve) {
            resolveClass(c);
        }
    }
}
```

```

        return c;
    }
}

```

在注释 1 处用来检查传入的类是否已经加载，如果已经加载则后面的代码不会执行，最后会返回该加载类。没有加载会接着向下执行，在注释 2 处，如果父类加载器不为 null，就调用父类加载器的 loadClass 方法。如果父类加载器为 null 就调用注释 3 处的 findBootstrapClassOrNull 方法，这个方法内部调用了 Native 方法 findBootstrapClass，findBootstrapClass 方法中最终会用 Bootstrap Classloader 来检查该类是否已经加载，如果没有加载就说明向上委托流程中没有加载该类，则调用注释 4 处的 findClass 方法继续向下进行查找流程。

采取双亲委托模式主要有如下两点好处。

- 避免重复加载，如果已经加载过一次 Class，就不需要再次加载，而是直接读取已经加载的 Class。
- 更加安全，如果不使用双亲委托模式，就可以自定义一个 String 类来替代系统的 String 类，这显然会造成安全隐患，采用双亲委托模式会使得系统的 String 类在 Java 虚拟机启动时就被加载，也就无法自定义 String 类来替代系统的 String 类，除非我们修改类加载器搜索类的默认算法。还有一点，只有两个类名一致并且被同一个类加载器加载的类，Java 虚拟机才会认为它们是同一个类，想要骗过 Java 虚拟机显然不会那么容易。

12.1.4 自定义ClassLoader

系统提供的类加载器只能够加载指定目录下的 jar 包和 Class 文件，如果想要加载网络上的或者 D 盘某一文件中的 jar 包和 Class 文件则需要自定义 ClassLoader。实现自定义 ClassLoader 需要如下两个步骤：

- (1) 定义一个自定义 ClassLoade 并继承抽象类 ClassLoader。
- (2) 复写 findClass 方法，并在 findClass 方法中调用 defineClass 方法。

下面我们就自定义一个 ClassLoader 用来加载位于 D:\lib 的 Class 文件。首先编写测试类并生成 Class 文件，如下所示：

```

package com.example;
public class Jobs {
    public void say() {

```

```

        System.out.println("One more thing");
    }
}

```

将这个 Jobs.java 放入到 D:\lib 中,使用 cmd 命令进入 D:\lib 目录中,执行 Javac Jobs.java 对该 java 文件进行编译,这时会在 D:\lib 中生成 Jobs.class。接下来在 AS 中创建一个 Java Library, 编写自定义 ClassLoader, 如下所示:

```

import java.io.ByteArrayOutputStream;
import java.io.File;
import java.io.FileInputStream;
import java.io.IOException;
import java.io.InputStream;
public class DiskClassLoader extends ClassLoader {
    private String path;
    public DiskClassLoader(String path) {
        this.path = path;
    }
    @Override
    protected Class<?> findClass(String name) throws ClassNotFoundException {
        Class clazz = null;
        byte[] classData = loadClassData(name);//1
        if (classData == null) {
            throw new ClassNotFoundException();
        } else {
            clazz= defineClass(name, classData, 0, classData.length);//2
        }
        return clazz;
    }
    private byte[] loadClassData(String name) {
        String fileName = getFileName(name);
        File file = new File(path,fileName);
        InputStream in=null;
        ByteArrayOutputStream out=null;
        try {
            in = new FileInputStream(file);
            out = new ByteArrayOutputStream();
            byte[] buffer = new byte[1024];
            int length=0;
            while ((length = in.read(buffer)) != -1) {
                out.write(buffer, 0, length);
            }
            return out.toByteArray();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}

```

```

    }finally {
        try {
            if(in!=null) {
                in.close();
            }
        } catch (IOException e) {
            e.printStackTrace();
        }
        try{
            if(out!=null) {
                out.close();
            }
        }catch (IOException e){
            e.printStackTrace();
        }
    }
    return null;
}
private String getFileName(String name) {
    int index = name.lastIndexOf('.');
    if(index == -1){//如果没有找到'.'则直接在末尾添加.class
        return name+".class";
    }else{
        return name.substring(index+1)+".class";
    }
}
}
}

```

这段代码有几点需要注意的，注释 1 处的 loadClassData 方法会获得 class 文件的字节码数组，并在注释 2 处调用 defineClass 方法将 class 文件的字节码数组转为 Class 类的实例。在 loadClassData 方法中需要对流进行操作，关闭流的操作要放在 finally 语句块中，并且要对 in 和 out 分别使用 try 语句，如果 in 和 out 共同在一个 try 语句中，假设 in.close() 发生异常的话，就无法执行 out.close()。最后我们来验证 DiskClassLoader 是否可用，代码如下所示：

```

import java.lang.reflect.InvocationTargetException;
import java.lang.reflect.Method;
public class ClassLoaderTest {
    public static void main(String[] args) {
        DiskClassLoader diskClassLoader = new DiskClassLoader("D:\\lib");//1
        try {
            Class c = diskClassLoader.loadClass("com.example.Jobs");//2
            if (c != null) {
                try {
                    Object obj = c.newInstance();

```

```

        System.out.println(obj.getClass().getClassLoader());
        Method method = c.getDeclaredMethod("say", null);
        method.invoke(obj, null);//3
    } catch (InstantiationException | IllegalAccessException
            | NoSuchMethodException
            | SecurityException |
            IllegalArgumentException |
            InvocationTargetException e) {
        e.printStackTrace();
    }
}
} catch (ClassNotFoundException e) {
    e.printStackTrace();
}
}
}

```

在注释 1 处创建 DiskClassLoader 并传入要加载类的路径,在注释 2 处加载 Class 文件,需要注意的是,不要在项目工程中存在名为 com.example.Jobs 的 Java 文件,否则就不会使用 DiskClassLoader 来加载,而是使用 AppClassLoader 来负责加载,这样我们定义的 DiskClassLoader 就变得毫无意义。接下来在注释 3 处通过反射来调用 Jobs 的 say 方法,打印结果如下:

```
com.example.DiskClassLoader@4554617c One more thing
```

使用了 DiskClassLoader 来加载 Class 文件,say 方法也正确执行,我们的目的就达到了。

12.2 Android中的ClassLoader

在 12.1 节我们学习了 Java 中的 ClassLoader,有的读者会把 Android 和 Java 中的 ClassLoader 搞混,甚至会认为 Java 中的 ClassLoader 和 Android 中的 ClassLoader 是一样的,这显然是不对的。这一篇文章我们就来学习 Android 中的 ClassLoader,来查看它和 Java 中的 ClassLoader 有何不同。

12.2.1 ClassLoader的类型

我们知道 Java 中的 ClassLoader 可以加载 jar 文件和 Class 文件(本质是加载 Class 文件),这一点在 Android 中并不适用,因为无论是 DVM 还是 ART,它们加载的不再是 Class

文件，而是 dex 文件，这就需要重新设计 ClassLoader 相关类，我们先来学习 ClassLoader 的类型。

Android 中的 ClassLoader 类型和 Java 中的 ClassLoader 类型类似，也分为两种类型，分别是系统类加载器和自定义加载器。其中系统类加载器主要包括 3 种，分别是 BootClassLoader、PathClassLoader 和 DexClassLoader。

1. BootClassLoader

Android 系统启动时会使用 BootClassLoader 来预加载常用类，与 SDK 中的 Bootstrap ClassLoader 不同，它并不是由 C/C++ 代码实现的，而是由 Java 实现的，BootClassLoader 的代码如下所示：

libcore/ojuni/src/main/java/java/lang/ClassLoader.java

```
class BootClassLoader extends ClassLoader {
    private static BootClassLoader instance;
    @FindBugsSuppressWarnings("DP_CREATE_CLASSLOADER_INSIDE_DO_PRIVILEGED")
    public static synchronized BootClassLoader getInstance() {
        if (instance == null) {
            instance = new BootClassLoader();
        }
        return instance;
    }
    ...
}
```

BootClassLoader 是 ClassLoader 的内部类，并继承自 ClassLoader。BootClassLoader 是一个单例类，需要注意的是 BootClassLoader 的访问修饰符是默认的，只有在同一个包中才可以访问，因此我们在应用程序中是无法直接调用的。

2. DexClassLoader

DexClassLoader 可以加载 dex 文件以及包含 dex 的压缩文件（apk 和 jar 文件），不管加载哪种文件，最终都要加载 dex 文件，在这一章为了方便理解和叙述，将 dex 文件以及包含 dex 的压缩文件统称为 dex 相关文件。查看 DexClassLoader 的代码，如下所示：

libcore/dalvik/src/main/java/dalvik/system/DexClassLoader.java

```
public class DexClassLoader extends BaseDexClassLoader {
    public DexClassLoader(String dexPath, String optimizedDirectory,
        String librarySearchPath, ClassLoader parent) {
```

```

        super(dexPath, new File(optimizedDirectory), librarySearchPath, parent);
    }
}

```

DexClassLoader 的构造方法有如下 4 个参数。

- dexPath: dex 相关文件路径集合, 多个路径用文件分隔符分隔, 默认文件分隔符为“:”。
- optimizedDirectory: 解压的 dex 文件存储路径, 这个路径必须是一个内部存储路径, 在一般情况下, 使用当前应用程序的私有路径: /data/data/<Package Name>/...。
- librarySearchPath: 包含 C/C++库的路径集合, 多个路径用文件分隔符分隔, 可以为 null。
- parent: 父加载器。

DexClassLoader 继承自 BaseDexClassLoader, 方法都在 BaseDexClassLoader 中实现。

3. PathClassLoader

Android 系统使用 PathClassLoader 来加载系统类和应用程序的类, 下面来查看它的代码:

```
libcore/dalvik/src/main/java/dalvik/system/PathClassLoader.java
```

```

public class PathClassLoader extends BaseDexClassLoader {
    public PathClassLoader(String dexPath, ClassLoader parent) {
        super(dexPath, null, null, parent);
    }
    public PathClassLoader(String dexPath, String librarySearchPath, ClassLoader
parent) {
        super(dexPath, null, librarySearchPath, parent);
    }
}

```

PathClassLoader 继承自 BaseDexClassLoader, 也都在 BaseDexClassLoader 中实现。

在 PathClassLoader 的构造方法中没有参数 optimizedDirectory, 这是因为 PathClassLoader 已经默认了参数 optimizedDirectory 的值为 /data/dalvik-cache, 很显然 PathClassLoader 无法定义解压的 dex 文件存储路径, 因此 PathClassLoader 通常用来加载已经安装的 apk 的 dex 文件 (安装的 apk 的 dex 文件会存储在 /data/dalvik-cache 中)。

12.2.2 ClassLoader 的继承关系

运行一个应用程序需要用到几种类型的类加载器呢? 如下所示:

```
public class MainActivity extends AppCompatActivity {
```

```

@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_main);
    ClassLoader loader = MainActivity.class.getClassLoader();
    while (loader != null) {
        Log.d("liuwangshu", loader.toString()); //1
        loader = loader.getParent();
    }
}
}

```

首先我们得到 MainActivity 的类加载器，并在注释 1 处通过 Log 打印出来，接着通过循环打印出当前类的类加载器的父加载器，直到没有父加载器终止循环，打印结果如下所示：

```

10-07 07:23:02.835 8272-8272/? D/liuwangshu:dalvik.system.PathClassLoader[DexPathList
[[zip file "/data/app/com.example.liuwangshu.moonclassloader-2/base.apk", zip file
"/data/app/com.example.liuwangshu.moonclassloader-2/split_lib_dependencies_apk.apk",
zip file "/data/app/com.example.liuwangshu.moonclassloader-2/split_lib_slice_0_apk.apk",
zip file "/data/app/com.example.liuwangshu.moonclassloader-2/split_lib_slice_1_apk.apk",
zip file "/data/app/com.example.liuwangshu.moonclassloader-2/split_lib_slice_2_apk.apk",
zip file "/data/app/com.example.liuwangshu.moonclassloader-2/split_lib_slice_3_apk.apk",
zip file "/data/app/com.example.liuwangshu.moonclassloader-2/split_lib_slice_4_apk.apk",
zip file "/data/app/com.example.liuwangshu.moonclassloader-2/split_lib_slice_5_apk.apk",
zip file "/data/app/com.example.liuwangshu.moonclassloader-2/split_lib_slice_6_apk.apk",
zip file "/data/app/com.example.liuwangshu.moonclassloader-2/split_lib_slice_7_apk.apk",
zip file "/data/app/com.example.liuwangshu.moonclassloader-2/split_lib_slice_8_apk.apk",
zip file "/data/app/com.example.liuwangshu.moonclassloader-2/split_lib_slice_9_apk.apk"],
nativeLibraryDirectories=[/data/app/com.example.liuwangshu.moonclassloader-2/lib/x86,
/vendor/lib, /system/lib]]] 10-07 07:23:02.835 8272-8272/?
D/liuwangshu:java.lang. BootClassLoader@e175998

```

可以看到有两种类加载器，一种是 PathClassLoader，另一种则是 BootClassLoader。DexPathList 中包含了很多 apk 的路径，其中 /data/app/com.example.liuwangshu.moonclassloader-2/base.apk 就是示例应用安装在手机上的位置。DexPathList 是在 BaseDexClassLoader 的构造方法中创建的，里面存储了 dex 相关文件的路径，在 ClassLoader 执行双亲委托模式的查找流程时会从 DexPathList 中进行查找，在 12.2.3 节我们还会再提到它。

除了上面所讲的 3 种主要的类加载器外，Android 还提供了其他的类加载器和 ClassLoader 相关类，ClassLoader 的继承关系如图 12-3 所示。

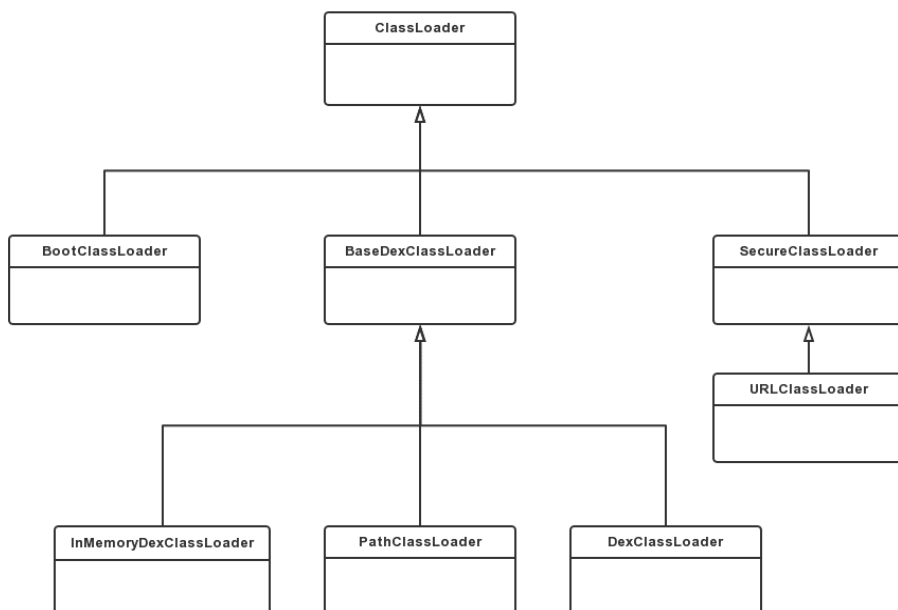


图 12-3 Android8.0 中 ClassLoader 的继承关系

可以看到图 12-3 中一共有 8 个 ClassLoader 相关类,其中有一些和 Java 中的 ClassLoader 相关类十分类似,下面简单对它们进行介绍:

- ClassLoader 是一个抽象类,其中定义了 ClassLoader 的主要功能。BootClassLoader 是它的内部类。
- SecureClassLoader 类和 JDK 8 中的 SecureClassLoader 类的代码是一样的,它继承了抽象类 ClassLoader。SecureClassLoader 并不是 ClassLoader 的实现类,而是拓展了 ClassLoader 类加入了权限方面的功能,加强了 ClassLoader 的安全性。
- URLClassLoader 类和 JDK 8 中的 URLClassLoader 类的代码是一样的,它继承自 SecureClassLoader,用来通过 URL 路径从 jar 文件和文件夹中加载类和资源。
- InMemoryDexClassLoader 是 Android 8.0 新增的类加载器,继承自 BaseDexClassLoader,用于加载内存中的 dex 文件。
- BaseDexClassLoader 继承自 ClassLoader,是抽象类 ClassLoader 的具体实现类,PathClassLoader、DexClassLoader 和 InMemoryDexClassLoader 都继承自它。

12.2.3 ClassLoader的加载过程

Android 的 ClassLoader 同样遵循了双亲委托模式,ClassLoader 的加载方法为 loadClass

方法，这个方法被定义在抽象类 `ClassLoader` 中，如下所示：

`libcore/ojrluni/src/main/java/java/lang/ClassLoader.java`

```
protected Class<?> loadClass(String name, boolean resolve)
throws ClassNotFoundException
{
    Class<?> c = findLoadedClass(name); //1
    if (c == null) {
        try {
            if (parent != null) { //2
                c = parent.loadClass(name, false);
            } else {
                c = findBootstrapClassOrNull(name); //3
            }
        } catch (ClassNotFoundException e) {
        }

        if (c == null) { //4
            c = findClass(name); //5
        }
    }
    return c;
}
```

`ClassLoader` 的 `loadClass` 方法和 12.1.3 节讲的 `loadClass` 方法（JDK 中 `ClassLoader` 的 `loadClass` 方法）类似。在注释 1 处用来检查传入的类是否已经加载，如果已经加载就返回该类，如果没有加载就在注释 2 处判断父加载器是否存在，存在就调用父加载器的 `loadClass` 方法，如果不存在就调用注释 3 处的 `findBootstrapClassOrNull` 方法，这个方法会直接返回 `null`。如果注释 4 处的代码成立，说明向上委托流程没有检查出类已经被加载，就会执行注释 5 处的 `findClass` 方法来进行查找流程，`findClass` 方法如下所示：

`libcore/ojrluni/src/main/java/java/lang/ClassLoader.java`

```
protected Class<?> findClass(String name) throws ClassNotFoundException {
    throw new ClassNotFoundException(name);
}
```

在 `findClass` 方法中直接抛出了异常，这说明 `findClass` 方法需要子类来实现，`BaseDexClassLoader` 的代码如下所示：

`libcore/dalvik/src/main/java/dalvik/system/BaseDexClassLoader.java`

```
...
public BaseDexClassLoader(String dexPath, File optimizedDirectory,
```

```

        String librarySearchPath, ClassLoader parent) {
    super(parent);
    this.pathList = new DexPathList(this, dexPath, librarySearchPath, null);
    if (reporter != null) {
        reporter.report(this.pathList.getDexPaths());
    }
}
...
@Override
protected Class<?> findClass(String name) throws ClassNotFoundException {
    List<Throwable> suppressedExceptions = new ArrayList<Throwable>();
    Class c = pathList.findClass(name, suppressedExceptions);//1
    if (c == null) {
        ClassNotFoundException cnfe = new ClassNotFoundException(
            "Didn't find class \"" + name + "\" on path: " + pathList);
        for (Throwable t : suppressedExceptions) {
            cnfe.addSuppressed(t);
        }
        throw cnfe;
    }
    return c;
}
...
}

```

在 BaseDexClassLoader 的构造方法中创建了 DexPathList，在注释 1 处调用了 DexPathList 的 findClass 方法：

libcore/dalvik/src/main/java/dalvik/system/DexPathList.java

```

public Class<?> findClass(String name, List<Throwable> suppressed) {
    for (Element element : dexElements) {//1
        Class<?> clazz = element.findClass(name, definingContext, suppressed);//2
        if (clazz != null) {
            return clazz;
        }
    }
    if (dexElementsSuppressedExceptions != null) {
        suppressed.addAll(Arrays.asList(dexElementsSuppressedExceptions));
    }
    return null;
}

```

在注释 1 处遍历 Element 数组 dexElements，在注释 2 处调用 Element 的 findClass 方法，Element 是 DexPathList 的静态内部类：

libcore/dalvik/src/main/java/dalvik/system/DexPathList.java

```

/*package*/ static class Element {
    private final File path;
    private final DexFile dexFile;
    private ClassPathURLStreamHandler urlHandler;
    private boolean initialized;
    public Element(DexFile dexFile, File dexZipPath) {
        this.dexFile = dexFile;
        this.path = dexZipPath;
    }
    public Element(DexFile dexFile) {
        this.dexFile = dexFile;
        this.path = null;
    }
    public Element(File path) {
        this.path = path;
        this.dexFile = null;
    }
    ...
    public Class<?> findClass(String name, ClassLoader definingContext,
        List<Throwable> suppressed) {
        return dexFile != null ? dexFile.loadClassBinaryName(name, defining
            Context, suppressed) : null;//1
    }
}

```

从 Element 的构造方法可以看出，其内部封装了 DexFile，它用于加载 dex。在注释 1 处如果 DexFile 不为 null 就调用 DexFile 的 loadClassBinaryName 方法：

libcore/dalvik/src/main/java/dalvik/system/DexFile.java

```

public Class loadClassBinaryName(String name, ClassLoader loader, List<Throwable>
    suppressed) {
    return defineClass(name, loader, mCookie, this, suppressed);
}

```

在 loadClassBinaryName 方法中调用了 defineClass 方法：

libcore/dalvik/src/main/java/dalvik/system/DexFile.java

```

private static Class defineClass(String name, ClassLoader loader, Object cookie,
    DexFile dexFile, List<Throwable> suppressed) {
    Class result = null;
    try {
        result = defineClassNative(name, loader, cookie, dexFile);//1
    }
}

```

```

    } catch (NoClassDefFoundError e) {
        if (suppressed != null) {
            suppressed.add(e);
        }
    }
    } catch (ClassNotFoundException e) {
        if (suppressed != null) {
            suppressed.add(e);
        }
    }
}
return result;
}

```

在注释 1 处调用了 `defineClassNative` 方法来加载 dex 相关文件，这个方法是 Native 方法，这里就不再进行分析，有兴趣的读者可以自行阅读源码。`ClassLoader` 的加载过程就是遵循着双亲委托模式，如果委托流程没有检查到此前加载过传入的类，就调用 `ClassLoader` 的 `findClass` 方法，Java 层最终会调用 `DexFile` 的 `defineClassNative` 方法来执行查找流程，如图 12-4 所示。

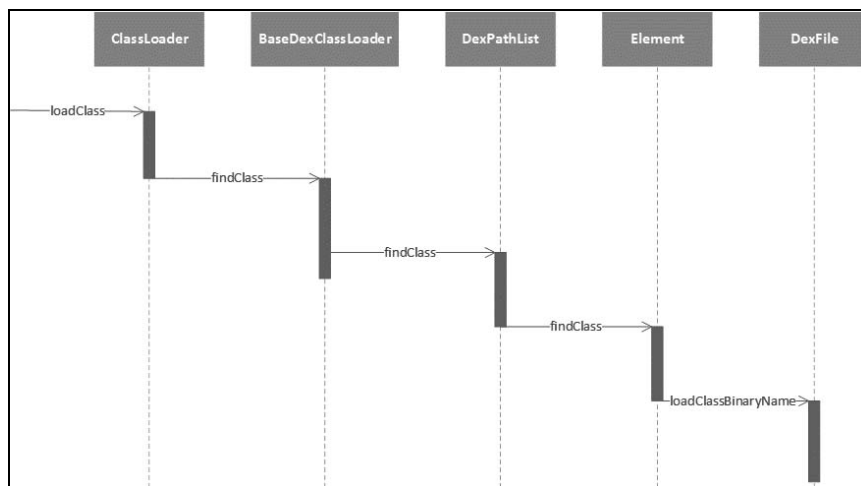


图 12-4 ClassLoader 查找流程

12.2.4 BootClassLoader 的创建

`BootClassLoader` 是在何时被创建的呢？这得先从 `Zygote` 进程开始说起，`ZygoteInit` 的 `main` 方法如下所示：

```

frameworks/base/core/java/com/android/internal/os/ZygoteInit.java
public static void main(String argv[]) {

```



```

...
    try {
        ...
        preload(bootTimingsTraceLog);
        ...
    }
}

```

main 方法是 ZygoteInit 的入口方法，其中调用了 ZygoteInit 的 preload 方法，在 preload 方法中又调用了 ZygoteInit 的 preloadClasses 方法，如下所示：

frameworks/base/core/java/com/android/internal/os/ZygoteInit.java

```

...
private static final String PRELOADED_CLASSES = "/system/etc/preloaded-classes";
...
private static void preloadClasses() {
    final VMRuntime runtime = VMRuntime.getRuntime();
    InputStream is;
    try {
        //将/system/etc/preloaded-classes 文件封装成 FileInputStream
        is = new FileInputStream(PRELOADED_CLASSES);//1
    } catch (FileNotFoundException e) {
        Log.e(TAG, "Couldn't find " + PRELOADED_CLASSES + ".");
        return;
    }
    ...
    try {
        //将FileInputStream 封装为 BufferedReader
        BufferedReader br
            = new BufferedReader(new InputStreamReader(is), 256);//2
        int count = 0;
        String line;
        while ((line = br.readLine()) != null) {//3
            line = line.trim();
            if (line.startsWith("#") || line.equals("")) {
                continue;
            }
            Trace.traceBegin(Trace.TRACE_TAG_DALVIK, line);
            try {
                if (false) {
                    Log.v(TAG, "Preloading " + line + "...");
                }
                Class.forName(line, true, null);//4
                count++;
            }
        }
    }
}

```

```

        } catch (ClassNotFoundException e) {
            Log.w(TAG, "Class not found for preloading: " + line);
        }
        ...
    } catch (IOException e) {
        Log.e(TAG, "Error reading " + PRELOADED_CLASSES + ".", e);
    } finally {
        ...
    }
}

```

preloadClasses 方法用于 Zygote 进程初始化时预加载常用类。在注释 1 处 PRELOADED_CLASSES 的值为将/system/etc/preloaded-classes 文件封装成 FileInputStream, preloaded-classes 文件中存有预加载类的目录, 这个文件在系统源码中的路径为 frameworks/base/preloaded-classes, 这里列举一些 preloaded-classes 文件中的预加载类名称, 如下所示:

```

android.app.ApplicationLoaders
android.app.ApplicationPackageManager
android.app.ApplicationPackageManager$OnPermissionsChangeListenerDelegate
android.app.ApplicationPackageManager$ResourceName
android.app.ContentProviderHolder
android.app.ContentProviderHolder$1
android.app.ContextImpl
android.app.ContextImpl$ApplicationContentResolver
android.app.DexLoadReporter
android.app.Dialog
android.app.Dialog$ListenersHandler
android.app.DownloadManager
android.app.Fragment

```

可以看到 preloaded-classes 文件中的预加载类的名称有很多都是我们非常熟知的。预加载属于拿空间换时间的策略, Zygote 环境配置得越健全越通用, 应用程序进程需要单独做的事情也就越少, 预加载除了预加载类, 还有预加载资源和预加载共享库, 因为不是本节的重点, 这里就不再延伸讲下去了。回到 preloadClasses 方法的注释 2 处, 将 FileInputStream 封装为 BufferedReader, 并在注释 3 处遍历 BufferedReader, 读出所有预加载类的名称, 每读出一个预加载类的名称就调用注释 4 处的代码加载该类, Class 的 forName 方法如下所示:

```
libcore/ojuni/src/main/java/java/lang/Class.java
```

```

@CallerSensitive
public static Class<?> forName(String name, boolean initialize,
ClassLoader loader)

```

```

        throws ClassNotFoundException
    {
        if (loader == null) {
            loader = BootClassLoader.getInstance();//1
        }
        Class<?> result;
        try {
            result = classForName(name, initialize, loader);//2
        } catch (ClassNotFoundException e) {
            Throwable cause = e.getCause();
            if (cause instanceof LinkageError) {
                throw (LinkageError) cause;
            }
            throw e;
        }
        return result;
    }
}

```

在注释 1 处创建了 BootClassLoader，并将 BootClassLoader 实例传入到了注释 2 处的 classForName 方法中，classForName 方法是 Native 方法，它的实现由 C/C++ 代码来完成，如下所示：

```

@FastNative
static native Class<?> classForName(String className, boolean shouldInitialize,
    ClassLoader classLoader) throws ClassNotFoundException;

```

Native 方法这里就不再分析了，我们知道了 BootClassLoader 是在 Zygote 进程的 Zygote 入口方法中被创建的，用于加载 preloaded-classes 文件中存有的预加载类。

12.2.5 PathClassLoader 的创建

PathClassLoader 的创建也得从 Zygote 进程开始说起，Zygote 进程启动 SystemServer 进程时会调用 ZygoteInit 的 startSystemServer 方法，如下所示：

```

frameworks/base/core/java/com/android/internal/os/ZygoteInit.java

private static boolean startSystemServer(String abiList, String socketName)
throws MethodAndArgsCaller, RuntimeException {
    ...
    int pid;
    try {
        parsedArgs = new ZygoteConnection.Arguments(args);
        ZygoteConnection.applyDebuggerSystemProperty(parsedArgs);
        ZygoteConnection.applyInvokeWithSystemProperty(parsedArgs);
    }
}

```

```

        /*1*/
        pid = Zygote.forkSystemServer(
            parsedArgs.uid, parsedArgs.gid,
            parsedArgs.gids,
            parsedArgs.debugFlags,
            null,
            parsedArgs.permittedCapabilities,
            parsedArgs.effectiveCapabilities);
    } catch (IllegalArgumentException ex) {
        throw new RuntimeException(ex);
    }
    if (pid == 0) { //2
        if (hasSecondZygote(abiList)) {
            waitForSecondaryZygote(socketName);
        }
        handleSystemServerProcess(parsedArgs); //3
    }
    return true;
}

```

在注释 1 处, Zygote 进程通过 forkSystemServer 方法 fork 自身创建子进程(SystemServer 进程)。在注释 2 处如果 forkSystemServer 方法返回的 pid 等于 0, 说明当前代码是在新创建的 SystemServer 进程中执行的, 接着就会执行注释 3 处的 handleSystemServerProcess 方法:

frameworks/base/core/java/com/android/internal/os/ZygoteInit.java

```

private static void handleSystemServerProcess(
    ZygoteConnection.Arguments parsedArgs)
    throws Zygote.MethodAndArgsCaller {
    ...
    if (parsedArgs.invokeWith != null) {
        ...
    } else {
        ClassLoader cl = null;
        if (systemServerClasspath != null) {
            cl = createPathClassLoader(systemServerClasspath, parsedArgs.
targetSdkVersion); //1
            Thread.currentThread().setContextClassLoader(cl);
        }
        ZygoteInit.zygoteInit(parsedArgs.targetSdkVersion, parsedArgs.
            remainingArgs, cl);
    }
}

```

在注释 1 处调用了 createPathClassLoader 方法, 如下所示:

```
frameworks/base/core/java/com/android/internal/os/ZygoteInit.java
```

```
static PathClassLoader createPathClassLoader(String classPath, int targetSdkVersion) {
    String libraryPath = System.getProperty("java.library.path");
    return PathClassLoaderFactory.createClassLoader(classPath, libraryPath,
        libraryPath, ClassLoader.getSystemClassLoader(), targetSdkVersion,
        true /* isNamespaceShared */);
}
```

在 `createPathClassLoader` 方法中又调用了 `PathClassLoaderFactory` 的 `createClassLoader` 方法，看来 `PathClassLoader` 是用工厂来进行创建的：

```
frameworks/base/core/java/com/android/internal/os/PathClassLoaderFactory.java
```

```
public static PathClassLoader createClassLoader(String dexPath,
    String librarySearchPath, String libraryPermittedPath, ClassLoader parent,
    int targetSdkVersion, boolean isNamespaceShared) {
    PathClassLoader pathClassLoader = new PathClassLoader(dexPath, librarySearchPath,
        parent);
    ...
    return pathClassLoader;
}
```

在 `PathClassLoaderFactory` 的 `createClassLoader` 方法中会创建 `PathClassLoader`。讲到这里可以得出结论，`PathClassLoader` 是在 `SystemServer` 进程中采用工厂模式创建的。

12.3 本章小结

本章分别对 Java 和 Android 的 `ClassLoader` 进行解析，通过 Java 和 Android 的 `ClassLoader` 的类型和 `ClassLoader` 的继承关系，就可以很清楚地看出它们的差异，主要有以下几点：

- Java 的引导类加载器是由 C++ 编写的，Android 中的引导类加载器则是用 Java 编写的。
- Android 的继承关系要比 Java 继承关系复杂一些，提供的功能也多。
- 由于 Android 中加载的不再是 Class 文件，因此 Android 中没有 `ExtClassLoader` 和 `AppClassLoader`，替代它们的是 `PathClassLoader` 和 `DexClassLoader`。

第 13 章

热修复原理

关联章节：第 9 章 JNI 原理；第 12 章 理解 ClassLoader

在 Android 应用开发中，热修复技术被越来越多的开发者所使用，也出现了很多热修复框架，比如 AndFix、Tinker、Dexposed 和 Nuwa 等。如果只是会这些热修复框架的使用那意义并不大，我们还需要了解它们的原理，这样不管热修复框架如何变化，只要基本原理不变，我们就可以很快地掌握它们。这一章不会对某些热修复框架源码进行解析，而是讲解热修复框架的通用原理。另外本章和第 12 章的内容联系紧密，阅读本章前最好先阅读第 12 章的内容。

13.1 热修复的产生

在开发过程中我们有可能遇到如下的情况。

- 刚发布的版本出现了严重的 Bug，这就需要去解决 Bug、测试并打包在各个应用市场上重新发布，这会耗费大量的人力和物力，代价会比较大。
- 已经改正了此前发布版本的 Bug，如果下一个版本是一个大版本，那么两个版本的间隔时间会很长，这样要等到下个大版本发布再修复 Bug，此前版本的 Bug 会长期地影响用户。
- 版本升级率不高，并且需要很长时间来完成版本覆盖，此前版本的 Bug 就会一直影响不升级版本的用户。

- 有一个小而重要的功能，需要短时间内完成版本覆盖，比如节日活动。

为了解决上面的问题，热修复框架就产生了。对于 Bug 的处理，开发人员不要过于依赖热修复框架，在开发的过程中还是要按照标准的流程做好自测，配合测试人员完成测试流程。

13.2 热修复框架的种类和对比

热修复框架的种类繁多，按照公司团队划分主要有如表 13-1 所示的几种。

表 13-1 按照公司团队划分热修复框架

类 别	成 员
阿里系	AndFix、Dexposed、阿里百川、Sophix
腾讯系	微信的 Tinker、QQ 空间的超级补丁、手机 QQ 的 QFix
知名公司	美团的 Robust、饿了么的 Amigo、美丽说蘑菇街的 Aceso
其他	RocooFix、Nuwa、AnoleFix

虽然热修复框架很多，但热修复框架的核心技术主要有三类，分别是代码修复、资源修复和动态链接库修复，其中每个核心技术又有很多不同的技术方案，每个技术方案又有不同的实现，另外这些热修复框架仍在不断地更新迭代中，可见热修复框架的技术实现是繁多可变的。作为开发者需要了解这些技术方案的基本原理，这样就可以以不变应万变。

部分热修复框架的对比如表 13-2 所示。

表 13-2 部分热修复框架的对比

特 性	AndFix	Tinker/Amigo	QQ 空间	Robust/Aceso
即时生效	是	否	否	是
方法替换	是	是	是	是
类替换	否	是	是	否
类结构修改	否	是	否	否
资源替换	否	是	是	否
so 替换	否	是	否	否
支持 gradle	否	是	否	否
支持 ART	是	是	是	是
支持 Android7.0	是	是	是	是

我们可以根据表 13-2 和具体业务来选择合适的热修复框架，当然表 13-2 所示的信息很难做到完全准确，因为部分的热修复框架还在不断更新迭代。

从表 13-2 中也可以发现 Tinker 和 Amigo 拥有的特性最多，是不是就选它们呢？也不尽然，拥有的特性多也意味着框架的代码量庞大，我们需要根据业务来选择最合适的，假设我们只是要用到方法替换，那么使用 Tinker 和 Amigo 显然是大材小用了。另外如果项目需要即时生效，那么使用 Tinker 和 Amigo 是无法满足需求的。对于即时生效，AndFix、Robust 和 Aceso 都满足这一点，这是因为 AndFix 的代码修复采用了底层替换方案，而 Robust 和 Aceso 的代码修复借鉴了 Instant Run 原理，这些原理后文会进行介绍。我们先来学习资源修复。

13.3 资源修复

很多热修复的框架的资源修复参考了 Instant Run 的资源修复的原理，因此我们首先要了解 Instant Run 是什么。

13.3.1 Instant Run概述

Instant Run 是 Android Studio 2.0 以后新增的一个运行机制，能够显著减少开发人员第二次及以后的构建和部署时间。在没有使用 Instant Run 前，我们编译部署应用程序的流程如图 13-1 所示。

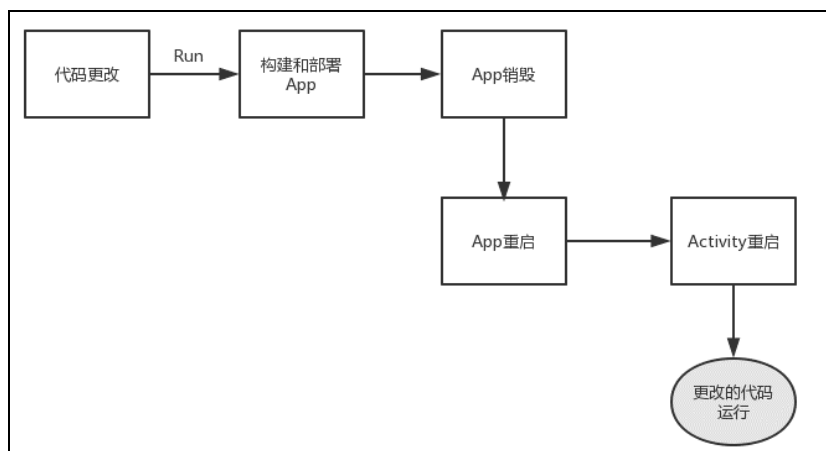


图 13-1 传统编译部署

从图 13-1 可以看出,传统的编译部署需要重新安装 App 和重启 App,这显然会很耗时,Instant Run 会避免这一情况,如图 13-2 所示。

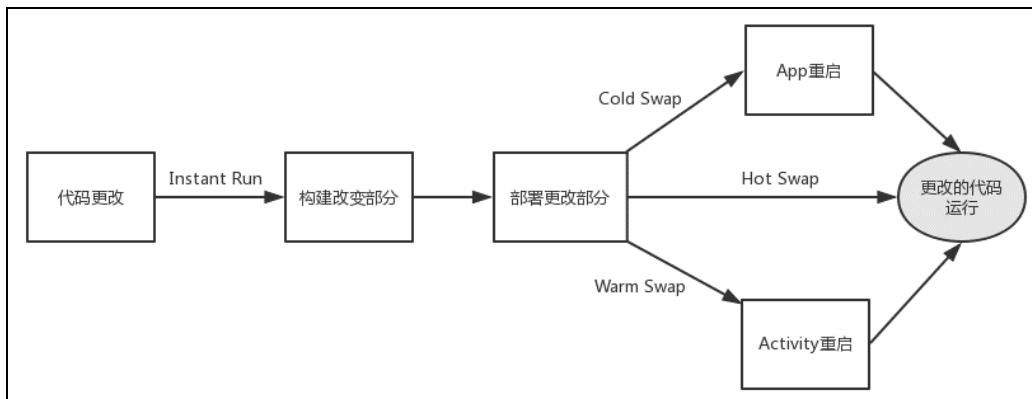


图 13-2 Instant Run 编译部署

从图 13-2 可以看出 Instant Run 的构建和部署都是基于更改的部分的。Instant Run 部署有三种方式，Instant Run 会根据代码的情况来决定采用哪种部署方式，无论哪种方式都不需要重新安装 App，这一点就已经提高了不少的效率。

- Hot swap: 从名称也可以看出 Hot Swap 是效率最高的部署方式，代码的增量改变不需要重启 App，甚至不需要重启当前的 Activity。修改一个现有方法中的代码时会采用 Hot Swap。
- Warm Swap: App 不需重启，但是 Activity 需要重启。修改或删除一个现有的资源文件时会采用 Warm Swap。
- Cold Swap: App 需要重启，但是不需要重新安装。采用 Cold Swap 的情况很多，比如添加、删除或修改一个字段和方法、添加一个类等。

13.3.2 Instant Run 的资源修复

既然很多热修复的框架资源修复参考了 Instant Run 的资源修复原理，那么我们了解 Instant Run 的资源修复原理就可以了。Instant Run 并不是 Android 的源码，需要通过反编译获取，可以参考相关书籍。

Instant Run 资源修复的核心逻辑在 MonkeyPatcher 的 monkeyPatchExistingResources 方法中，如下所示：

```
com/android/tools/fd/runtime/MonkeyPatcher.java
```

```
public static void monkeyPatchExistingResources(Context context,
String externalResourceFile, Collection<Activity> activities) {
    if (externalResourceFile == null) {
        return;
    }
    try {
        //创建一个新的 AssetManager
        AssetManager newAssetManager = (AssetManager)
AssetManager.class.getConstructor(new Class[0]).newInstance(new
Object[0]);//1
Method mAddAssetPath = AssetManager.class.getDeclaredMethod(
"addAssetPath", new Class[] { String.class });//2
        mAddAssetPath.setAccessible(true);
        //通过反射调用 addAssetPath 方法加载外部的资源(SD 卡)
        if (((Integer) mAddAssetPath.invoke(newAssetManager,new Object[] { external
ResourceFile })).intValue() == 0) {//3
            throw new IllegalStateException("Could not create new AssetManager");
        }
        Method mEnsureStringBlocks = AssetManager.class.getDeclaredMethod(
"ensureStringBlocks", new Class[0]);
        mEnsureStringBlocks.setAccessible(true);
        mEnsureStringBlocks.invoke(newAssetManager, new Object[0]);
        if (activities != null) {
            for (Activity activity : activities) {
                Resources resources = activity.getResources();//4
                try {
                    //反射得到 Resources 的 AssetManager 类型的 mAssets 字段
                    Field mAssets = Resources.class.getDeclaredField("mAssets");//5
                    mAssets.setAccessible(true);
                    //将 mAssets 字段的引用替换为新创建的 AssetManager
                    mAssets.set(resources, newAssetManager);//6
                } catch (Throwable ignore) {
                    ...
                }
                //得到 Activity 的 Resources.Theme
                Resources.Theme theme = activity.getTheme();
                try {
                    try {
                        //反射得到 Resources.Theme 的 mAssets 字段
                        Field ma = Resources.Theme.class.getDeclaredField("mAssets");
                        ma.setAccessible(true);
                        //将 Resources.Theme 的 mAssets 字段替换为 newAssetManager
                        ma.set(theme, newAssetManager);//7
                    }
                }
            }
        }
    }
}
```

```

    } catch (NoSuchFieldException ignore) {
        ...
    }
    ...
} catch (Throwable e) {
    Log.e("InstantRun", "Failed to update existing theme for activity " +
        activity, e);
}
pruneResourceCaches(resources);
    }
}
/**
 * 根据 SDK 版本的不同, 用不同的方式得到 Resources 的弱引用集合
 */
Collection<WeakReference<Resources>> references;
if (Build.VERSION.SDK_INT >= 19) {
    Class<?> resourcesManagerClass = Class.forName("android.app.Resources
    Manager");
    Method mGetInstance = resourcesManagerClass.getDeclaredMethod(
        "getInstance", new Class[0]);
    mGetInstance.setAccessible(true);
    Object resourcesManager = mGetInstance.invoke(null, new Object[0]);
    try {
        Field fMActiveResources = resourcesManagerClass.getDeclaredField
            ("mActiveResources");
        fMActiveResources.setAccessible(true);
        ArrayMap<?, WeakReference<Resources>> arrayMap
        = (ArrayMap) fMActiveResources.get(resourcesManager);
        references = arrayMap.values();
    } catch (NoSuchFieldException ignore) {
        Field mResourceReferences = resourcesManagerClass
            .getDeclaredField("mResourceReferences");
        mResourceReferences.setAccessible(true);
        references = (Collection) mResourceReferences.get(resourcesManager);
    }
} else {
    Class<?> activityThread = Class.forName("android.app.ActivityThread");
    Field fMActiveResources = activityThread.getDeclaredField("mActive
    Resources");
    fMActiveResources.setAccessible(true);
    Object thread = getActivityThread(context, activityThread);
    HashMap<?, WeakReference<Resources>> map = (HashMap) fMActiveResources
        .get(thread);
    references = map.values();
}

```

```

// 遍历并得到弱引用集中的 Resources, 将 Resources 的 mAssets 字段引用替换成新的 AssetManager
for (WeakReference<Resources> wr : references) {
    Resources resources = (Resources) wr.get();
    if (resources != null) {
        try {
            Field mAssets = Resources.class.getDeclaredField("mAssets");
            mAssets.setAccessible(true);
            mAssets.set(resources, newAssetManager);
        } catch (Throwable ignore) {
            ...
        }
        resources.updateConfiguration(resources.getConfiguration(),
            resources.getDisplayMetrics());
    }
} catch (Throwable e) {
    throw new IllegalStateException(e);
}
}

```

在注释 1 处创建一个新的 AssetManager, 在注释 2 和注释 3 处通过反射调用 addAssetPath 方法加载外部(SD 卡)的资源。在注释 4 处遍历 Activity 列表, 得到每个 Activity 的 Resources, 在注释 5 处通过反射得到 Resources 的 AssetManager 类型的 mAssets 字段, 并在注释 6 处改写 mAssets 字段的引用为新的 AssetManager。采用同样的方式, 在注释 7 处将 Resources.Theme 的 mAssets 字段的引用替换为新创建的 AssetManager。紧接着根据 SDK 版本的不同, 用不同的方式得到 Resources 的弱引用集合, 再遍历这个弱引用集合, 将弱引用集合中的 Resources 的 mAssets 字段引用都替换成新创建的 AssetManager。

可以看出 Instant Run 中的资源热修复可以简单地总结为两个步骤:

(1) 创建新的 AssetManager, 通过反射调用 addAssetPath 方法加载外部的资源, 这样新创建的 AssetManager 就含有了外部资源。

(2) 将 AssetManager 类型的 mAssets 字段的引用全部替换为新创建的 AssetManager。

13.4 代码修复

代码修复主要有 3 个方案, 分别是底层替换方案、类加载方案和 Instant Run 方案。

13.4.1 类加载方案

类加载方案基于 Dex 分包方案，什么是 Dex 分包方案呢？这个得先从 65536 限制和 LinearAlloc 限制说起。

1. 65536 限制

随着应用功能越来越复杂，代码量不断地增大，引入的库也越来越多，可能会在编译时提示如下异常：

```
com.android.dex.DexIndexOverflowException: method ID not in [0, 0xffff]: 65536
```

这说明应用中引用的方法数超过了最大数 65536 个。产生这一问题的原因就是系统的 65536 限制，65536 限制的主要原因是 DVM Bytecode 的限制，DVM 指令集的方法调用指令 invoke-kind 索引为 16bits，最多能引用 65535 个方法。

2. LinearAlloc 限制

在安装应用时可能会提示 INSTALL_FAILED_DEXOPT，产生的原因就是 LinearAlloc 限制，DVM 中的 LinearAlloc 是一个固定的缓存区，当方法数超出了缓存区的大小时会报错。

为了解决 65536 限制和 LinearAlloc 限制，从而产生了 Dex 分包方案。Dex 分包方案主要做的是在打包时将应用代码分成多个 Dex，将应用启动时必须用到的类和这些类的直接引用类放到主 Dex 中，其他代码放到次 Dex 中。当应用启动时先加载主 Dex，等到应用启动后再动态地加载次 Dex，从而缓解了主 Dex 的 65536 限制和 LinearAlloc 限制。

Dex 分包方案主要有两种，分别是 Google 官方方案、Dex 自动拆包和动态加载方案。因为 Dex 分包方案不是本章的重点，这里就不再过多的介绍，我们接着来学习类加载方案。在 12.2.3 节中学习了 ClassLoader 的加载过程，其中一个环节就是调用 DexPathList 的 findClass 的方法，如下所示：

libcore/dalvik/src/main/java/dalvik/system/DexPathList.java

```
public Class<?> findClass(String name, List<Throwable> suppressed) {
    for (Element element : dexElements) { //1
        Class<?> clazz = element.findClass(name, definingContext, suppressed); //2
        if (clazz != null) {
            return clazz;
        }
    }
    if (dexElementsSuppressedExceptions != null) {
        suppressed.addAll(Arrays.asList(dexElementsSuppressedExceptions));
    }
}
```

```

    }
    return null;
}

```

Element 内部封装了 DexFile，DexFile 用于加载 dex 文件，因此每个 dex 文件对应一个 Element。多个 Element 组成了有序的 Element 数组 dexElements。当要查找类时，会在注释 1 处遍历 Element 数组 dexElements（相当于遍历 dex 文件数组），注释 2 处调用 Element 的 findClass 方法，其方法内部会调用 DexFile 的 loadClassBinaryName 方法查找类。如果在 Element 中（dex 文件）找到了该类就返回，如果没有找到就接着在下一个 Element 中进行查找。根据上面的查找流程，我们将有 Bug 的类 Key.class 进行修改，再将 Key.class 打包成包含 dex 的补丁包 Patch.jar，放在 Element 数组 dexElements 的第一个元素，这样会首先找到 Patch.dex 中的 Key.class 去替换之前存在 Bug 的 Key.class，排在数组后面的 dex 文件中存在 Bug 的 Key.class 根据 ClassLoader 的双亲委托模式就不会被加载，这就是类加载方案，如图 13-3 所示。

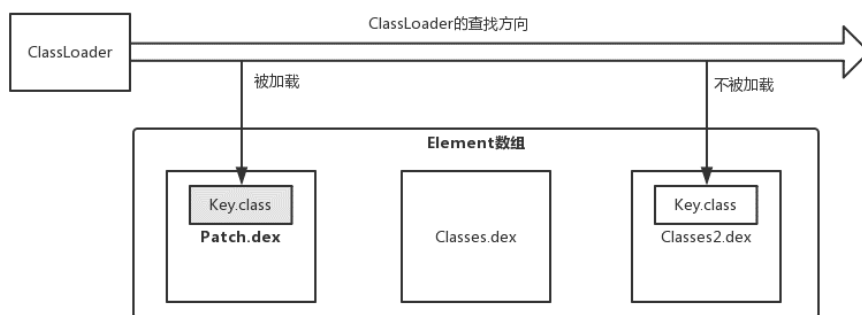


图 13-3 类加载方案

类加载方案需要重启 App 后让 ClassLoader 重新加载新的类，为什么需要重启呢？这是因为类是无法被卸载的，要想重新加载新的类就需要重启 App，因此采用类加载方案的热修复框架是不能即时生效的。虽然很多热修复框架采用了类加载方案，但具体的实现细节和步骤还是有一些区别的，比如 QQ 空间的超级补丁和 Nuwa 是按照上面说的将补丁包放在 Element 数组的第一个元素得到优先加载。微信 Tinker 将新旧 APK 做了 diff，得到 patch.dex，再将 patch.dex 与手机中 APK 的 classes.dex 做合并，生成新的 classes.dex，然后在运行时通过反射将 classes.dex 放在 Element 数组的第一个元素。饿了么的 Amigo 则是将补丁包中每个 dex 对应的 Element 取出来，之后组成新的 Element 数组，在运行时通过反射用新的 Element 数组替换掉现有的 Element 数组。

采用类加载方案的主要是以腾讯系为主，包括微信的 Tinker、QQ 空间的超级补丁、手机 QQ 的 QFix、饿了么的 Amigo 和 Nuwa 等。

13.4.2 底层替换方案

与类加载方案不同的是，底层替换方案不会再次加载新类，而是直接在 Native 层修改原有类，由于在原有类进行修改限制会比较多，且不能增减原有类的方法和字段，如果我们增加了方法数，那么方法索引数也会增加，这样访问方法时会无法通过索引找到正确的方法，同样的字段也是类似的情况。底层替换方案和反射的原理有些关联，就拿方法替换来说，方法反射我们可以调用 `java.lang.Class.getDeclaredMethod`，假设我们要反射 `Key` 的 `show` 方法，会调用如下所示的代码：

```
Key.class.getDeclaredMethod("show").invoke(Key.class.newInstance());
```

Android 8.0 的 `invoke` 方法，如下所示：

```
libcore/ojuni/src/main/java/java/lang/reflect/Method.java
```

```
@FastNative
public native Object invoke(Object obj, Object... args)
    throws IllegalAccessException, IllegalArgumentException, Invocation
TargetException;
```

`invoke` 方法是一个 native 方法，对于 Jni 层的代码为：

```
art/runtime/native/java_lang_reflect_Method.cc
```

```
static jobject Method_invoke(JNIEnv* env, jobject javaMethod, jobject javaReceiver,
    jobject javaArgs) {
    ScopedFastNativeObjectAccess soa(env);
    return InvokeMethod(soa, javaMethod, javaReceiver, javaArgs);
}
```

在 `Method_invoke` 函数中又调用了 `InvokeMethod` 函数：

```
art/runtime/reflection.cc
```

```
jobject InvokeMethod(const ScopedObjectAccessAlreadyRunnable& soa, jobject
javaMethod, jobject javaReceiver, jobject javaArgs, size_t num_frames) {
...
    ObjPtr<mirror::Executable> executable = soa.Decode<mirror::Executable>(javaMethod);
    const bool accessible = executable->IsAccessible();
    ArtMethod* m = executable->GetArtMethod();//1
...
}
```

注释 1 处获取传入的 javaMethod (Key 的 show 方法) 在 ART 虚拟机中对应的一个 ArtMethod 指针, ArtMethod 结构体中包含了 Java 方法的所有信息, 包括执行入口、访问权限、所属类和代码执行地址等, ArtMethod 结构如下所示:

```
art/runtime/art_method.h
```

```
class ArtMethod FINAL {
...
protected:
    GcRoot<mirror::Class> declaring_class_;
    std::atomic<std::uint32_t> access_flags_;
    uint32_t dex_code_item_offset_;
    uint32_t dex_method_index_;
    uint16_t method_index_;
    uint16_t hotness_count_;
    struct PtrSizedFields {
        ArtMethod** dex_cache_resolved_methods_;//1
        void* data_;
        void* entry_point_from_quick_compiled_code_;//2
    } ptr_sized_fields_;
}
```

在 ArtMethod 结构中比较重要的字段是注释 1 处的 dex_cache_resolved_methods 和注释 2 处的 entry_point_from_quick_compiled_code, 它们是方法的执行入口, 当我们调用某一个方法时 (比如 Key 的 show 方法), 就会取得 show 方法的执行入口, 通过执行入口就可以跳过去执行 show 方法。替换 ArtMethod 结构体中的字段或者替换整个 ArtMethod 结构体, 这就是底层替换方案。AndFix 采用的是替换 ArtMethod 结构体中的字段, 这样会有兼容问题, 因为厂商可能会修改 ArtMethod 结构体, 导致方法替换失败。Sophix 采用的是替换整个 ArtMethod 结构体, 这样不会存在兼容问题。底层替换方案直接替换了方法, 可以立即生效不需要重启。采用底层替换方案主要是阿里系为主, 包括 AndFix、Dexposed、阿里百川、Sophix。

13.4.3 Instant Run方案

除了资源修复, 代码修复同样也可以借鉴 Instant Run 的原理, 可以说 Instant Run 的出现推动了热修复框架的发展。Instant Run 在第一次构建 APK 时, 使用 ASM 在每一个方法中注入了类似如下的代码:

```
IncrementalChange localIncrementalChange = $change;//1
if (localIncrementalChange != null) {//2
```



```

        localIncrementalChange.access$dispatch(
            "onCreate.(Landroid/os/Bundle;)V", new Object[] { this,
                paramBundle });
        return;
    }

```

其中注释 1 处是一个成员变量 `localIncrementalChange`，它的值为 `$change`，`$change` 实现了 `IncrementalChange` 这个抽象接口。当我们点击 `InstantRun` 时，如果方法没有变化则 `$change` 为 `null`，就调用 `return`，不做任何处理。如果方法有变化，就生成替换类，这里我们假设 `MainActivity` 的 `onCreate` 方法做了修改，就会生成替换类 `MainActivity$Override`，这个类实现了 `IncrementalChange` 接口，同时也会生成一个 `AppPatchesLoaderImpl` 类，这个类的 `getPatchedClasses` 方法会返回被修改的类的列表（里面包含了 `MainActivity`），根据列表会将 `MainActivity` 的 `$change` 设置为 `MainActivity$Override`，因此满足了注释 2 的条件，会执行 `MainActivity$Override` 的 `access$dispatch` 方法，在 `access$dispatch` 方法中会根据参数 `"onCreate.(Landroid/os/Bundle;)V"` 执行 `MainActivity$Override` 的 `onCreate` 方法，从而实现了 `onCreate` 方法的修改。借鉴 `Instant Run` 的原理的热修复框架有 `Robust` 和 `Aceso`。

什么是 ASM?

ASM 是一个 Java 字节码操控框架，它能够动态生成类或者增强现有类的功能。ASM 可以直接产生 `class` 文件，也可以在类被加载到虚拟机之前动态改变类的行为。

13.5 动态链接库的修复

Android 平台的动态链接库主要指的是 `so` 库，为了更好地理解，本章动态链接库简称为 `so`。热修复框架的 `so` 的修复的主要是更新 `so`，换句话说就是重新加载 `so`，因此 `so` 的修复的基础原理就是加载 `so`。

13.5.1 System的load和loadLibrary方法

加载 `so` 主要用到了 `System` 类的 `load` 和 `loadLibrary` 方法，如下所示：

```

libcore/ojuni/src/main/java/java/lang/System.java

public final class System {
    ...
    @CallerSensitive
    public static void load(String filename) {

```

```

        Runtime.getRuntime().load0(VMStack.getStackClass1(), filename);//1
    }
    @CallerSensitive
    public static void loadLibrary(String libname) {
        Runtime.getRuntime().loadLibrary0(VMStack.getCallingClassLoader(),
            libname);//2
    }
    ...
}

```

System 的 load 方法传入的参数是 so 在磁盘的完整路径, 用于加载指定路径的 so。System 的 loadLibrary 方法传入的参数是 so 的名称, 用于加载 App 安装后自动从 apk 包中复制到 /data/data/packageName/lib 下的 so。目前 so 的修复都是基于这两个方法, 这里分别对这两个方法进行讲解。

1. System 的 load 方法

注释 1 处的 Runtime.getRuntime() 会得到当前 Java 应用程序的运行环境 Runtime, Runtime 的 load0 方法如下所示:

libcore/ojuni/src/main/java/java/lang/Runtime.java

```

synchronized void load0(Class<?> fromClass, String filename) {
    if (!(new File(filename).isAbsolute())) {
        throw new UnsatisfiedLinkError(
            "Expecting an absolute path of the library: " + filename);
    }
    if (filename == null) {
        throw new NullPointerException("filename == null");
    }
    String error = doLoad(filename, fromClass.getClassLoader());//1
    if (error != null) {
        throw new UnsatisfiedLinkError(error);
    }
}

```

在注释 1 处调用了 doLoad 方法, 并将加载该类的类加载器作为参数传入进去:

libcore/ojuni/src/main/java/java/lang/Runtime.java

```

private String doLoad(String name, ClassLoader loader) {
    String librarySearchPath = null;
    if (loader != null && loader instanceof BaseDexClassLoader) {
        BaseDexClassLoader dexClassLoader = (BaseDexClassLoader) loader;
        librarySearchPath = dexClassLoader.getLdLibraryPath();
    }
}

```

```

    }
    synchronized (this) {
        return nativeLoad(name, loader, librarySearchPath);
    }
}

```

doLoad 方法会调用 native 方法 nativeLoad，关于 nativeLoad 方法后面会讲到。

2. System 的 loadLibrary 方法

我们接着来查看 System 的 loadLibrary 方法，其中会调用 Runtime 的 loadLibrary0 方法：

libcore/ojuni/src/main/java/java/lang/Runtime.java

```

synchronized void loadLibrary0(ClassLoader loader, String libname) {
    if (libname.indexOf((int)File.separatorChar) != -1) {
        throw new UnsatisfiedLinkError(
            "Directory separator should not appear in library name: " + libname);
    }
    String libraryName = libname;
    if (loader != null) {
        String filename = loader.findLibrary(libraryName);//1
        if (filename == null) {
            throw new UnsatisfiedLinkError(loader + " couldn't find \"" +
                System.mapLibraryName(libraryName) + "\"");
        }
        String error = doLoad(filename, loader);//2
        if (error != null) {
            throw new UnsatisfiedLinkError(error);
        }
        return;
    }

    String filename = System.mapLibraryName(libraryName);
    List<String> candidates = new ArrayList<String>();
    String lastError = null;
    for (String directory : getLibPaths()) {//3
        String candidate = directory + filename;//4
        candidates.add(candidate);
        if (IoUtils.canOpenReadOnly(candidate)) {
            String error = doLoad(candidate, loader);//5
            if (error == null) {
                return; // We successfully loaded the library. Job done.
            }
            lastError = error;
        }
    }
}

```

```

    }
    if (lastError != null) {
        throw new UnsatisfiedLinkError(lastError);
    }
    throw new UnsatisfiedLinkError("Library " + libraryName + " not found; tried "
        + candidates);
}

```

loadLibrary0 方法分为两个部分，一个是传入的 ClassLoader 不为 null 的部分，另一个是 ClassLoader 为 null 的部分，我们先来看 ClassLoader 为 null 的部分。在注释 3 处遍历 getLibPaths 方法，这个方法会返回 java.library.path 选项配置的路径数组。在注释 4 处拼接出 so 路径并传入注释 5 处调用的 doLoad 方法中。当 ClassLoader 不为 null 时，在注释 2 处同样调用了 doLoad 方法，其中第一个参数是通过注释 1 处的 ClassLoader 的 findLibrary 方法来得到的，findLibrary 方法在 ClassLoader 的实现类 BaseDexClassLoader 中实现。

libcore/dalvik/src/main/java/dalvik/system/BaseDexClassLoader.java

```

@Override
public String findLibrary(String name) {
    return pathList.findLibrary(name);
}

```

在 findLibrary 方法中调用了 DexPathList 的 findLibrary 方法：

libcore/dalvik/src/main/java/dalvik/system/DexPathList.java

```

public String findLibrary(String libraryName) {
    String fileName = System.mapLibraryName(libraryName);
    for (NativeLibraryElement element : nativeLibraryPathElements) {
        String path = element.findNativeLibrary(fileName); //1
        if (path != null) {
            return path;
        }
    }
    return null;
}

```

这和 13.3.1 节讲到的 DexPathList 的 findClass 方法类似，在 NativeLibraryElement 数组中的每一个 NativeLibraryElement 对应一个 so 库，在注释 1 处调用 NativeLibraryElement 的 findNativeLibrary 方法就可以返回 so 的路径。上面的代码结合 13.3.1 节的类加载方案，就可以得到 so 的修复的一种方案，就是将 so 补丁插入到 NativeLibraryElement 数组的前部，让 so 补丁的路径先被返回，并调用 Runtime 的 doLoad 方法进行加载，在 doLoad 方法中会

调用 native 方法 nativeLoad。看来 System 的 load 方法和 loadLibrary 方法在 Java Framework 层最终调用的都是 nativeLoad 方法。我们接着来分析 nativeLoad 方法。

13.5.2 nativeLoad 方法分析

nativeLoad 方法对应的 JNI 层函数如下所示：

libcore/ojuni/src/main/native/Runtime.c

```
JNIEXPORT jstring JNICALL
Runtime_nativeLoad(JNIEnv* env, jclass ignored, jstring javaFilename,
                  jobject javaLoader, jstring javaLibrarySearchPath)
{
    return JVM_NativeLoad(env, javaFilename, javaLoader, javaLibrarySearchPath);
}
```

在 Runtime_nativeLoad 函数中调用了 JVM_NativeLoad 函数：

art/runtime/openjdkjvm/OpenjdkJvm.cc

```
JNIEXPORT jstring JVM_NativeLoad(JNIEnv* env,
                                  jstring javaFilename,
                                  jobject javaLoader,
                                  jstring javaLibrarySearchPath) {
    //将 so 的文件名转换为 ScopedUtfChars 类型
    ScopedUtfChars filename(env, javaFilename);
    if (filename.c_str() == NULL) {
        return NULL;
    }
    std::string error_msg;
    {
        //获取当前运行时的虚拟机
        art::JavaVMExt* vm = art::Runtime::Current()->GetJavaVM();//1
        //虚拟机加载 so
        bool success = vm->LoadNativeLibrary(env,
                                              filename.c_str(),
                                              javaLoader,
                                              javaLibrarySearchPath,
                                              &error_msg);

        if (success) {
            return nullptr;
        }
    }
    env->ExceptionClear();
    return env->NewStringUTF(error_msg.c_str());
}
```

在注释 1 处获取当前运行时的 JavaVMExt 类型指针，JavaVMExt 用于代表一个虚拟机实例，紧接着调用 JavaVMExt 的 LoadNativeLibrary 函数来加载 so。LoadNativeLibrary 函数代码有些多，这里分为 3 个部分来讲解。

1. LoadNativeLibrary 函数 part1

art/runtime/java_vm_ext.cc

```
bool JavaVMExt::LoadNativeLibrary(JNIEnv* env,
                                   const std::string& path,
                                   jobject class_loader,
                                   jstring library_path,
                                   std::string* error_msg) {
    error_msg->clear();
    SharedLibrary* library;
    Thread* self = Thread::Current();
    {
        // TODO: move the locking (and more of this logic) into Libraries.
        MutexLock mu(self, *Locks::jni_libraries_lock_);
        library = libraries_->Get(path); //1
    }
    ...
    if (library != nullptr) { //2
        if (library->GetClassLoaderAllocator() != class_loader_allocator) { //3
            StringAppendF(error_msg, "Shared library \"%s\" already opened by "
                                "ClassLoader %p; can't open in ClassLoader %p",
                                path.c_str(), library->GetClassLoader(), class_loader);
            LOG(WARNING) << error_msg;
            return false;
        }
        VLOG(jni) << "[Shared library \"" << path << "\" already loaded in "
                                << " ClassLoader " << class_loader << "];"
        if (!library->CheckOnLoadResult()) { //4
            StringAppendF(error_msg, "JNI_OnLoad failed on a previous attempt "
                                "to load \"%s\"", path.c_str());
            return false;
        }
        return true;
    }
}
```

在注释 1 处根据 so 的名称从 libraries_ 中获取对应的 SharedLibrary 类型指针 library，如果满足注释 2 处的条件就说明此前加载过该 so。在注释 3 处如果此前加载用的 ClassLoader 和当前传入的 ClassLoader 不相同的话，就会返回 false，在注释 4 处判断上次加载 so 的结

果，如果有异常也会返回 false，中断 so 加载。如果满足了注释 2、注释 3、注释 4 处的条件就会返回 true，不再重复加载 so。

2. LoadNativeLibrary 函数 part2

```
...
Locks::mutator_lock_>AssertNotHeld(self);
const char* path_str = path.empty() ? nullptr : path.c_str();
bool needs_native_bridge = false;
/**
 * 1 打开路径 path_str 的 so 库，得到 so 句柄 handle
 */
void* handle = android::OpenNativeLibrary(env,
                                           runtime_>GetTargetSdkVersion(),
                                           path_str,
                                           class_loader,
                                           library_path,
                                           &needs_native_bridge,
                                           error_msg);

VLOG(jni) << "[Call to dlopen(\"" << path << "\", RTLD_NOW) returned " << handle
<< " ]";
if (handle == nullptr) { //2
    VLOG(jni) << "dlopen(\"" << path << "\", RTLD_NOW) failed: " << *error_msg;
    return false;
}

if (env->ExceptionCheck() == JNI_TRUE) {
    LOG(ERROR) << "Unexpected exception:";
    env->ExceptionDescribe();
    env->ExceptionClear();
}
bool created_library = false;
{
    /**
     * 3 创建 SharedLibrary
     */
    std::unique_ptr<SharedLibrary> new_library(
        new SharedLibrary(env,
                          self,
                          path,
                          handle,
                          needs_native_bridge,
```

```

        class_loader,
        class_loader_allocator));

MutexLock mu(self, *Locks::jni_libraries_lock_);
library = libraries_>Get(path); //4
if (library == nullptr) { // We won race to get libraries_lock.
    library = new_library.release();
    libraries_>Put(path, library);
    created_library = true;
}
}
...

```

在注释 1 处根据 so 的路径 path_str 来打开该 so，并返回得到 so 句柄，在注释 2 处如果获取 so 句柄失败就会返回 false，中断 so 加载。在注释 3 处新创建 SharedLibrary，并将 so 句柄作为参数传入进去。在注释 4 处获取传入 path 对应的 library，如果 library 为空指针，就将新创建的 SharedLibrary 赋值给 library，并将 library 存储到 libraries_ 中。

3. LoadNativeLibrary 函数 part3

```

...
bool was_successful = false;
void* sym = library->FindSymbol("JNI_OnLoad", nullptr); //1
if (sym == nullptr) { //2
    VLOG(jni) << "[No JNI_OnLoad found in \"" << path << "\"]";
    was_successful = true;
} else {
    ScopedLocalRef<jobject> old_class_loader(env, env->NewLocalRef(self->
        GetClassLoaderOverride()));
    self->SetClassLoaderOverride(class_loader);
    VLOG(jni) << "[Calling JNI_OnLoad in \"" << path << "\]";
    typedef int (*JNI_OnLoadFn)(JavaVM*, void*);
    JNI_OnLoadFn jni_on_load = reinterpret_cast<JNI_OnLoadFn>(sym);
    int version = (*jni_on_load)(this, nullptr); //3
    if (runtime_->GetTargetSdkVersion() != 0 && runtime_->GetTargetSdkVersion() <=
        21) {
        EnsureFrontOfChain(SIGSEGV);
    }
    self->SetClassLoaderOverride(old_class_loader.get());
    if (version == JNI_ERR) {
        StringAppendF(error_msg, "JNI_ERR returned from JNI_OnLoad in \"%s\"",
            path.c_str());
    } else if (JavaVMExt::IsBadJniVersion(version)) {

```



```

        StringAppendF(error_msg, "Bad JNI version returned from JNI_OnLoad in \"%s\": %d",
                        path.c_str(), version);
    } else {
        was_successful = true; //4
    }
    VLOG(jni) << "[Returned " << (was_successful ? "successfully" : "failure")
                << " from JNI_OnLoad in \"" << path << "\"]";
}
library->SetResult(was_successful);
return was_successful;
}

```

在注释 1 处查找 JNI_OnLoad 函数的指针并赋值给空指针 sym，在 9.2.3 节中我们知道 JNI_OnLoad 函数用于 native 方法的动态注册。在注释 2 处如果没有找到 JNI_OnLoad 函数就将 was_successful 赋值为 true，说明已经加载成功，没有找到 JNI_OnLoad 函数也算加载成功，这是因为并不是所有 so 都定义了 JNI_OnLoad 函数，因为 native 方法除了动态注册，还有静态注册。如果找到了 JNI_OnLoad 函数，就在注释 3 处执行 JNI_OnLoad 函数并将结果赋值给 version，如果 version 为 JNI_ERR 或者 BadJniVersion，说明没有执行成功，was_successful 的值仍旧为默认 false，否则就将 was_successful 赋值为 true，最终会返回该 was_successful。

4. LoadNativeLibrary 函数总结

LoadNativeLibrary 函数的行数很多，这里来做一个总结，LoadNativeLibrary 函数主要做了如下 3 方面工作。

(1) 判断 so 是否被加载过，两次 ClassLoader 是否是同一个，避免 so 重复加载。

(2) 打开 so 并得到 so 句柄，如果 so 句柄获取失败，就返回 false。创建新的 SharedLibrary，如果传入 path 对应的 library 为空指针，就将新创建的 SharedLibrary 赋值给 library，并将 library 存储到 libraries_ 中。

(3) 查找 JNI_OnLoad 的函数指针，根据不同情况设置 was_successful 的值，最终返回该 was_successful。

这样总结可能有些抽象，LoadNativeLibrary 函数的流程图如图 13-4 所示。

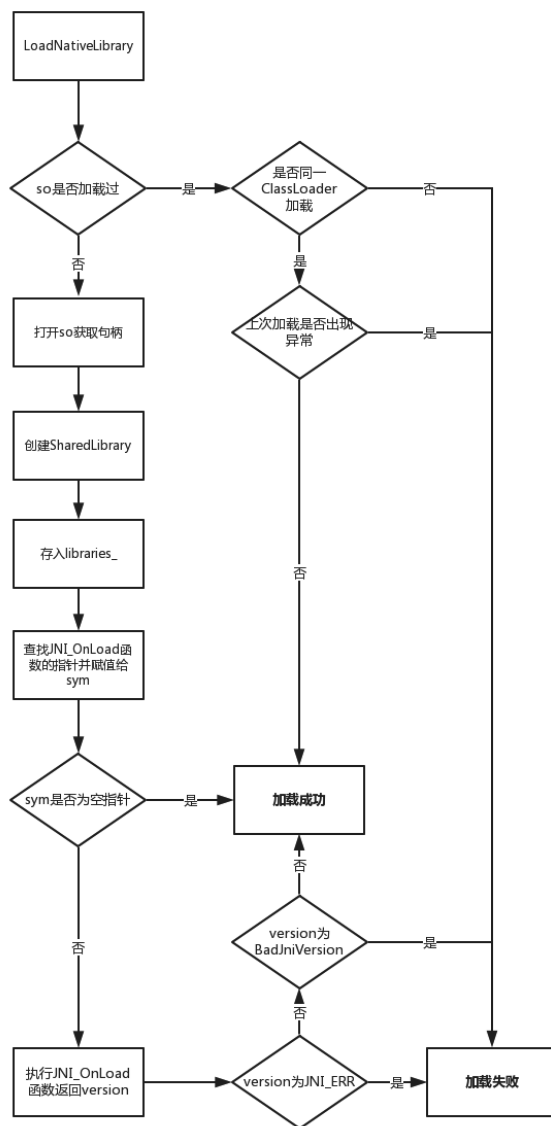


图 13-4 LoadNativeLibrary 函数的流程图

讲到这里总结一下 so 修复主要有两个方案：

(1) 将 so 补丁插入到 NativeLibraryElement 数组的前部，让 so 补丁的路径先被返回和加载。

(2) 调用 System 的 load 方法来接管 so 的加载入口。

13.6 本章小结

本章没有像《Android 进阶之光》一样直接去介绍这些框架的源码，主要有两点原因：一个是热修复的框架太多了；另一个就是这些框架还在不断地更新迭代中。因此本章直接从热修复的原理角度入手，目前所有的热修复框架都是基于这些原理开发的，我们掌握这些原理就可以以不变应万变，一方面可以更好地理解热修复框架，应用于开发之中；另一方面对阅读热修复框架的源码起到了铺垫作用。

第 14 章

Hook 技术

关联章节：第 4 章 四大组件的工作过程

本章主要的目的是为下一章讲插件化原理做铺垫，插件化涉及的技术非常多，比如应用程序启动流程、四大组件启动流程、AMS 原理、ClassLoader 原理、上下文 Context 等，这些技术在本书前面部分已经介绍过，还有很多技术由于书的篇幅原因没有介绍，比如包管理机制、Gradle、Binder 机制等，除此之外还有一个技术是必须介绍的，这就是 Hook 技术，可见它在插件化技术中的重要性，Hook 技术的知识点非常多，本章只涉及插件化相关的部分。

说到 Hook 技术得先提到逆向工程，逆向工程源于商业及军事领域中的硬件分析，其主要目的是在不能轻易获得必要的生产信息的情况下，直接从成品分析，推导出产品的设计原理。逆向分析分为静态分析和动态分析，其中静态分析指的是一种在不执行程序的情况下对程序行为进行分析的技术；动态分析是指在程序运行时对程序进行调试的技术。Hook 技术就属于动态分析，它不仅在 Android 平台中被应用，早在 Windows 平台中就已经被应用了。

14.1 Hook 技术概述

我们知道 Android 系统的代码调用和回调都是按照一定顺序执行的，这里举一个简单

的例子，如图 14-1 所示。

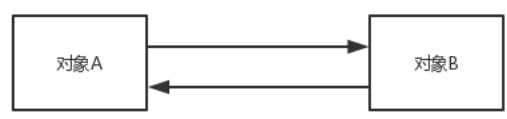


图 14-1 正常的调用和回调

对象 A 调用类对象，对象 B 处理后将数据回调给对象 A，接着来看采用 Hook 的调用流程，如图 14-2 所示。

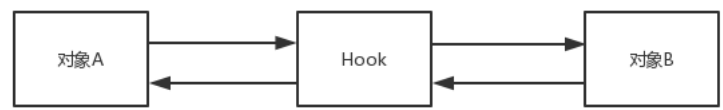


图 14-2 Hook 的调用和回调

图 14-2 中的 Hook 可以是一个方法或者一个对象，它像一个钩子一样挂在对象 A 和对象 B 之间，当对象 A 调用对象 B 之前做一些处理（比如修改方法的参数和返回值），起到了“欺上瞒下”的作用，与其说 Hook 是钩子，不如说是劫持来的更贴切些。我们知道应用程序进程之间是彼此独立的，应用程序进程和系统进程之间也是如此，想要在应用程序进程更改系统进程的某些行为很难直接实现，有了 Hook 技术，我们就可以在进程间进行行为更改，如图 14-3 所示。

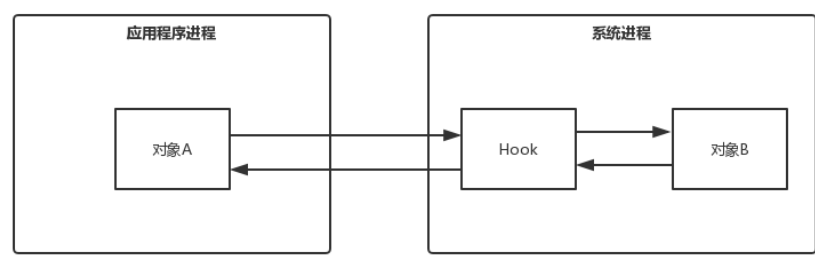


图 14-3 Hook

可以看到 Hook 可以将自己融入到它所劫持的对象（对象 B）所在的进程中，成为系统进程的一部分，这样我们就可以通过 Hook 来更改对象 B 的行为。被劫持的对象（对象 B），称作 Hook 点，为了保证 Hook 的稳定性，Hook 点一般选择容易找到并且不易变化的对象，静态变量和单例就符合这一条件。

14.2 Hook技术分类

Hook 技术知识点比较多，因此 Hook 技术根据不同的角度会有很多种分类，这里介绍其中的三种分类。

根据 Hook 的 API 语言划分，分为 Hook Java 和 Hook Native。

- Hook Java 主要通过反射和代理来实现，应用于在 SDK 开发环境中修改 Java 代码。
- Hook Native 则应用于在 NDK 开发环境和系统开发中修改 Native 代码。

根据 Hook 的进程划分，分为应用程序进程 Hook 和全局 Hook。

- 应用程序进程 Hook 只能 Hook 当前所在的应用程序进程。
- 应用程序进程是 Zygote 进程 fork 出来的，如果对 Zygote 进行 Hook，就可以实现 Hook 系统所有的应用程序进程，这就是全局 Hook。

根据 Hook 的实现方式划分，分为如下两种。

- 通过反射和代理实现，只能 Hook 当前的应用程序进程。
- 通过 Hook 框架来实现，比如 Xposed，可以实现全局 Hook，但是需要 root。

Hook Native、全局 Hook 和通过 Hook 框架实现这些分类和插件化技术关联不大，本章主要需要学习的是 Hook Java，想要更好地学习 Hook Java，首先要了解代理模式。

14.3 代理模式

代理模式也叫委托模式，是结构型设计模式的一种。在现实生活中我们用到类似代理模式的场景有很多，比如代购、代理上网、打官司等。

定义：为其他对象提供一种代理以控制对这个对象的访问称为代理模式。

代理模式结构图如图 14-4 所示。

在代理模式中有如下角色。

- Subject：抽象主题类，声明真实主题与代理的共同接口方法。
- RealSubject：真实主题类，定义了代理所表示的集体对象，客户端通过代理类间接调用真实主题类的方法。

- Proxy: 代理类, 持有对真实主题类的引用, 在其所实现的接口方法中调用真实主题类中相应的接口方法执行。
- Client: 客户端类。

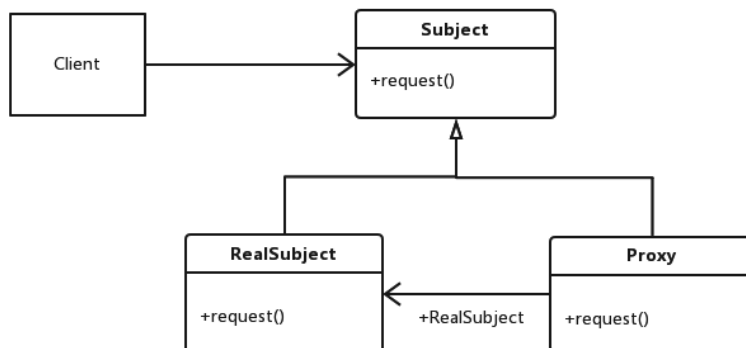


图 14-4 代理模式结构图

14.3.1 代理模式简单实现

假设我要买哈尔滨的红肠, 但是工作忙也没时间回哈尔滨, 就委托在哈尔滨的朋友帮我购买。

1. 抽象主题类

抽象主题类具有真实主题类和代理的共同接口方法, 我想要代购, 那共同的方法就是购买:

```
public interface IShop {
    void buy();
}
```

2. 真实主题类

这个购买者 LiuWangShu 也就是我, 实现了 IShop 接口提供的 buy()方法, 如下所示:

```
public class LiuWangShu implements IShop {
    @Override
    public void buy() {
        System.out.println("购买");
    }
}
```

3. 代理类

我找的朋友就是代理类，同样也需要实现 IShop 接口，并且要持有被代理者，在 buy 方法中调用了被代理者的 buy 方法：

```
public class Purchasing implements IShop {
    private IShop mShop;
    public Purchasing(IShop shop){
        mShop=shop;
    }
    @Override
    public void buy() {
        mShop.buy();
    }
}
```

4. 客户端类

```
public class Client {
    public static void main(String[] args){
        //创建 LiuWangShu
        IShop liuwangshu=new LiuWangShu();
        //创建代购者并将 LiuWangShu 作为构造函数参数传入
        IShop purchasing=new Purchasing(liuwangshu);
        purchasing.buy();
    }
}
```

客户端类的代码就是代理类包含了真实主题类（被代理者），最终调用的都是真实主题类（被代理者）实现的方法，在上面的例子就是 LiuWangShu 类的 buy 方法，所以运行的结果就是“购买”。

14.3.2 动态代理的简单实现

从编码的角度来说，代理模式分为静态代理和动态代理，14.2.1 节的例子是静态代理，在代码运行前就已经存在了代理类的 class 编译文件，而动态代理则是在代码运行时通过反射来动态地生成代理类的对象，并确定到底来代理谁。也就是我们在编码阶段不需要知道代理谁，代理谁我们将在代码运行时决定。Java 提供了动态的代理接口 InvocationHandler，实现该接口需要重写 invoke 方法。下面我们在上面静态代理的例子上做修改，首先创建动态代理类，代码如下所示：


```

public class DynamicPurchasing implements InvocationHandler{
    private Object obj;
    public DynamicPurchasing(Object obj){
        this.obj=obj;
    }
    @Override
    public Object invoke(Object proxy, Method method, Object[] args) throws Throwable {
        Object result=method.invoke(obj, args);
        if(method.getName().equals("buy")){
            System.out.println("Liuwangshu 在买买买");
        }
        return result;
    }
}

```

在动态代理类中我们声明一个 Object 的引用，该引用指向被代理类，我们调用被代理类的具体方法在 invoke 方法中执行。接下来我们修改客户端类代码：

```

public class Client {
    public static void main(String[] args){
        //创建 LiuWangShu
        IShop liuwangshu=new LiuWangShu();
        //创建动态代理
        DynamicPurchasing mDynamicPurchasing=new DynamicPurchasing(liuwangshu);
        //创建 LiuWangShu 的 ClassLoader
        ClassLoader loader=liuwangshu.getClass().getClassLoader();
        //动态创建代理类
        IShop purchasing= (IShop) Proxy.newProxyInstance(loader,new Class[]{IShop.class},mDynamicPurchasing);
        purchasing.buy();
    }
}

```

调用 Proxy.newProxyInstance()来生成动态代理类，调用 purchasing 的 buy 方法会调用 DynamicPurchasing 的 invoke 方法。代理模式从编码的角度可以分为静态代理和动态代理。

14.4 Hook startActivity方法

我们知道 Hook 可以用来劫持对象，被劫持的对象叫作 Hook 点，用代理对象来替代 Hook 点，这样我们就可以在代理上实现自己想做的操作。这里以 Hook 常用的 startActivity

方法来举例，startActivity 方法分为两个，如下所示：

```
startActivity(intent);
getApplicationContext().startActivity(intent);
```

第一个是 Activity 的 startActivity 方法，第二个是 Context 的 startActivity 方法，这两个方法的调用链是不同的，这里分开进行讲解。

14.4.1 Hook Activity的startActivity方法

Activity 的 startActivity 方法在 4.1.1 节中提到过，代码如下所示：

frameworks/base/core/java/android/app/Activity.java

```
@Override
public void startActivity(Intent intent) {
    this.startActivity(intent, null);
}
```

又调用了 startActivity 方法：

frameworks/base/core/java/android/app/Activity.java

```
@Override
public void startActivity(Intent intent, @Nullable Bundle options) {
    if (options != null) {
        startActivityForResult(intent, -1, options);
    } else {
        startActivityForResult(intent, -1);
    }
}
```

接着查看 startActivityForResult 方法，如下所示：

frameworks/base/core/java/android/app/Activity.java

```
@Override
public void startActivityForResult(
    String who, Intent intent, int requestCode, @Nullable Bundle options) {
    Uri referrer = onProvideReferrer();
    if (referrer != null) {
        intent.putExtra(Intent.EXTRA_REFERRER, referrer);
    }
    options = transferSpringboardActivityOptions(options);
    Instrumentation.ActivityResult ar =
        mInstrumentation.execStartActivity(
            this, mMainThread.getApplicationThread(), mToken, who,
```

```

        intent, requestCode, options); //1
    if (ar != null) {
        mMainThread.sendActivityResult(
            mToken, who, requestCode,
            ar.getResultCode(), ar.getResultData());
    }
    cancelInputsAndStartExitTransition(options);
}

```

在注释 1 处调用了 `mInstrumentation` 的 `execStartActivity` 方法来启动 Activity, 剩余的调用代码已经在 4.1 节介绍了, 这里不再赘述。这个 `mInstrumentation` 是 Activity 的成员变量, 我们就选择 Instrumentation 为 Hook 点, 用代理 Instrumentation 来替代原始的 Instrumentation 来完成 Hook。首先我们先写出代理 Instrumentation 类, 如下所示:

InstrumentationProxy.java

```

public class InstrumentationProxy extends Instrumentation {
    private static final String TAG = "InstrumentationProxy";
    Instrumentation mInstrumentation;
    public InstrumentationProxy(Instrumentation instrumentation) {
        mInstrumentation = instrumentation;
    }
    public ActivityResult execStartActivity(
        Context who, IBinder contextThread, IBinder token, Activity target,
        Intent intent, int requestCode, Bundle options) {
        Log.d(TAG, "Hook 成功" + "---who:" + who);
        try {
            //通过反射找到 Instrumentation 的 execStartActivity 方法
            Method execStartActivity = Instrumentation.class.getDeclaredMethod(
                "execStartActivity",
                Context.class, IBinder.class, IBinder.class, Activity.class,
                Intent.class, int.class, Bundle.class);
            return (ActivityResult) execStartActivity.invoke(mInstrumentation, who,
                contextThread, token, target, intent, requestCode, options);
        } catch (Exception e) {
            throw new RuntimeException(e);
        }
    }
}

```

InstrumentationProxy 继承了 Instrumentation, 并包含 Instrumentation 的引用。InstrumentationProxy 实现了 `execStartActivity` 方法, 其内部会通过反射找到并调用 Instrumentation 的 `execStartActivity` 方法。接下来我们用 InstrumentationProxy 来替换 Instrumentation, 代码如下所示:

MainActivity.java

```

public void replaceActivityInstrumentation(Activity activity){
    try {
        //得到 Activity 的 mInstrumentation 字段
        Field field = Activity.class.getDeclaredField("mInstrumentation");//1
        //取消 Java 的权限控制检查
        field.setAccessible(true);//2
        Instrumentation instrumentation = (Instrumentation)field.get(activity);//3
        Instrumentation instrumentationProxy = new InstrumentationProxy
            (instrumentation);//4
        field.set(activity,instrumentationProxy);
    } catch (Exception e){
        e.printStackTrace();
    }
}

```

在注释 1 处得到 Activity 的成员变量 mInstrumentation，这个成员变量是私用的，因此在注释 2 处取消 Java 的权限控制检查，这样就可以访问 mInstrumentation 字段。在注释 3 处得到 Activity 中的 Instrumentation 对象，在注释 4 处创建 InstrumentationProxy 并传入注释 3 处得到的 Instrumentation 对象，最后用 InstrumentationProxy 来替换 Instrumentation，这样就实现了代理 Instrumentation 替换 Instrumentation 的目的。最后在 MainActivity 的 onCreate 方法中使用 replaceActivityInstrumentation 方法，如下所示：

MainActivity.java

```

public class MainActivity extends Activity {
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        replaceActivityInstrumentation(this);
        Intent intent = new Intent(Intent.ACTION_VIEW);
        intent.setData(Uri.parse("http://liuwangshu.cn"));
        startActivity(intent);
    }
    ...
}

```

在 onCreate 方法中访问了我的独立博客，打印的 Log 如下：

```

D/InstrumentationProxy: Hook 成功---who:com.example.liuwangshu.hookinstrumentation.
MainActivity@47fcd35

```

14.4.2 Hook Context的startActivity方法

在第 5 章讲过, Context 的实现类为 ContextImpl, ContextImpl 的 startActivity 方法如下所示:

frameworks/base/core/java/android/app/ContextImpl.java

```
@Override
public void startActivity(Intent intent, Bundle options) {
    warnIfCallingFromSystemProcess();
    if ((intent.getFlags() & Intent.FLAG_ACTIVITY_NEW_TASK) == 0
        && options != null && ActivityOptions.fromBundle(options).getLaunch
        TaskId() == -1) {
        throw new AndroidRuntimeException(
            "Calling startActivity() from outside of an Activity "
            + " context requires the FLAG_ACTIVITY_NEW_TASK flag."
            + " Is this really what you want?");
    }
    mMainThread.getInstrumentation().execStartActivity(
        getOuterContext(), mMainThread.getApplicationThread(), null,
        (Activity) null, intent, -1, options);
}
```

最后一行调用了 ActivityThread 的 getInstrumentation 方法获取 Instrumentation。ActivityThread 是主线程的管理类, Instrumentation 是 ActivityThread 的成员变量, 一个进程中只有一个 ActivityThread, 因此仍旧选择 Instrumentation 作为 Hook 点, 用代理 Instrumentation 来替换 Instrumentation。代理 Instrumentation 和 14.3.1 节给出的 InstrumentationProxy 代码是一样的, 接下来我们用 InstrumentationProxy 来替换 Instrumentation, 代码如下所示:

MainActivity.java

```
public void replaceContextInstrumentation(){
    try {
        //获取 ActivityThread 类
        Class<?> activityThreadClazz = Class.forName("android.app.ActivityThread");
        Field activityThreadField=activityThreadClazz.getDeclaredField
        ("sCurrentActivityThread");//1
        activityThreadField.setAccessible(true);
        Object currentActivityThread= activityThreadField.get(null);//2
        //获取 ActivityThread 中的 mInstrumentation 字段
        Field mInstrumentationField = activityThreadClazz.getDeclaredField
        ("mInstrumentation");
        mInstrumentationField.setAccessible(true);
```

```

        Instrumentation mInstrumentation = (Instrumentation) mInstrumentation
        Field.get(currentActivityThread);
        Instrumentation mInstrumentationProxy = new InstrumentationProxy
        (mInstrumentation);//3
        mInstrumentationField.set(currentActivityThread, mInstrumentationProxy);
    } catch (Exception e) {
        e.printStackTrace();
    }
}

```

首先我们通过反射获取 ActivityThread 类，ActivityThread 类中有一个静态变量 sCurrentActivityThread，用于表示当前的 ActivityThread 对象，因此在注释 1 处获取 ActivityThread 中定义的 sCurrentActivityThread 字段，在注释 2 处获取 Field 类型的 activityThreadField 对象的值，这个值就是 sCurrentActivityThread 对象。同理获取当前 ActivityThread 的 mInstrumentation 对象。在注释 3 处创建 InstrumentationProxy 并传入此前得到的 mInstrumentation 对象，最后用 InstrumentationProxy 来替换 mInstrumentation。在 MainActivity 中使用 replaceContextInstrumentation 方法，如下所示：

MainActivity.java

```

public class MainActivity extends Activity {
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        replaceContextInstrumentation();
        Intent intent = new Intent(Intent.ACTION_VIEW);
        intent.setData(Uri.parse("http://liuwangshu.cn"));
        intent.addFlags(Intent.FLAG_ACTIVITY_NEW_TASK);
        getApplicationContext().startActivity(intent);
    }
}

```

在 onCreate 方法中仍旧访问了我的独立博客，打印的 Log 为：

```
D/InstrumentationProxy: Hook 成功---who:android.app.Application@2f21e30
```

14.4.3 Hook startActivity 总结

Hook Context 的 startActivity 方法和 Hook Activity 的 startActivity 方法最大的区别就是替换的 Instrumentation 不同，前者是 ActivityThread 中的 Instrumentation，后者是 Activity 中的 Instrumentation。另外有一点需要注意的是，这里举的例子都是在 MainActivity 的 onCreate 方法中进行 Instrumentation 替换的，这里未必是最佳的替换时间点，只是为了方

便举例。可以尝试在 Activity 的 `attachBaseContext` 方法中进行 Instrumentation 替换，因为这个方法要先于 Activity 的 `onCreate` 方法被调用。讲到这里，我们知道了如何使用代理来 Hook `startActivity` 方法，简单说就是找到 Hook 点，再用代理对象来替换 Hook 点。

14.5 本章小结

本章原本是下一章的内容，但是为了讲解得更清晰，所以就单拿出来作为一章。本章简单地介绍了 Hook 技术概述、Hook 技术分类和 Hook `startActivity` 方法，这些内容主要是为了下一章讲解插件化做铺垫，因此并没有讲解 Hook 技术的全部内容，想要更多地了解 Hook 技术请查阅相关专业书籍。

第 15 章

插件化原理

关联章节：第 2 章 Android 系统启动；第 4 章 四大组件的工作过程；
第 6 章 理解 ActivityManagerService；第 9 章 JNI 原理；
第 12 章 理解 ClassLoader；第 13 章 热修复原理；第 14 章 Hook 技术

随着应用开发的规模和复杂度越来越高，插件化技术被广泛地应用在各个较大规模的应用开发中。插件化技术和热修复技术都属于动态加载技术，从普及率的角度来看，插件化技术没有热修复的普及率高，主要原因是占大多数的中小型应用很少也没有必要去采用插件化技术。虽然插件化技术普及率现在不算高，但是插件化的原理对于应用开发的技术提升有很大的帮助，可以使你更好地理解系统的源码，并将系统源码和应用开发相结合。插件化是一个很庞大的知识体系，用一章的内容只能介绍部分的插件化原理，本章更多的是起一个抛砖引玉的作用。在本书截稿之前，Android P preview 开始限制调用隐藏 API，很快也出现了一些绕过限制的方案，但无论采用什么方案，插件化的基本原理还是需要去了解的。阅读本章前请先阅读开头列出的关联章节，以达到最好的阅读理解效果。

15.1 动态加载技术

在讲到插件化原理之前，需要先了解它的前身：动态加载技术。动态加载技术不只应用在 Android 开发领域，在很多开发领域都应用过，这里我们讨论的只是动态加载技术在 Android 开发领域的应用。在 Android 传统开发中，一旦应用的代码被打包成 APK 并被上

传到各个渠道市场，我们就不能修改应用的源码了，只能通过服务器来控制应用中预留的分支代码。但是很多时候我们无法提前预知需求和突然发生的情况，也就不能提前在应用代码中预留分支代码，这时就需要采用动态加载技术。在应用程序运行时，动态加载一些程序中原本不存在的可执行文件并运行这些文件里的代码逻辑。可执行文件总的来说分为两种，一种是动态链接库 so，另一种是 dex 相关文件（dex 以及包含 dex 的 jar/apk 文件）。看到这里很多读者可能发现了，在第 13 章我们讲解热修复原理时也提到了上述的可执行文件的加载，这是因为热修复技术本身就是动态加载技术派生出来的。随着应用开发技术和业务的逐步发展，动态加载技术派生出两个技术，分别是热修复技术和插件化技术，如图 15-1 所示。

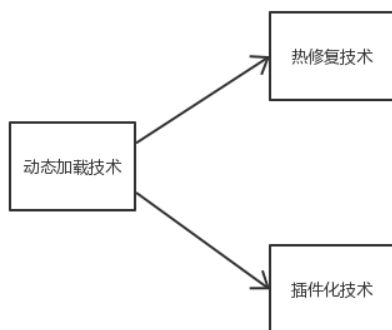


图 15-1 动态加载技术的派生技术

其中热修复技术主要用来修复 Bug，插件化技术则主要用于解决应用越来越庞大以及功能模块的解耦，围绕着两个技术出现了很多的热修复框架和插件化框架，在第 13 章我们了解了很多热修复框架，这一章我们将了解一些插件化框架。需要注意的是，动态加载技术本身并没有被官方认可，并且是一个非常规的技术，在国外这门技术关注度并不高，它的产生更多的是国内的业务需求和产品的驱动。

15.2 插件化的产生

在讲解插件化原理之前，我们十分有必要了解插件化是如何产生的。

15.2.1 应用开发的痛点和瓶颈

在 Android 开发早期很少用到动态加载技术，因为这个时候业务需求和应用开发的复

杂度都不是很高，但随着互联网的极速发展，会出现以下几种情况。

1. 业务复杂，模块耦合

随着业务越来越复杂也越来越多，导致应用程序体积越来越大，应用程序的工程和功能模块数量越来越多，一个应用可能是由几十、几百人协同开发的，很多工程和功能模块都是由一个小组进行开发维护的，如果功能模块间的耦合度比较高，修改一个功能模块会影响其他功能模块，势必会极大地增加沟通成本。

2. 应用间的接入

一个应用不再是单独的应用，它可能需要接入其他的应用。拿手机淘宝来说，它的流量非常大，其他的淘宝应用或者业务比如：聚划算、淘宝书城、飞猪旅游、淘宝拍卖、淘宝外卖（口碑外卖）等都希望接入到淘宝客户端，这样既能获取到流量，同时也可以将用户引流到自己的应用中，如果使用常规的技术手段，会产生两个问题。

- 比如淘宝外卖需要接入到淘宝客户端中，那么淘宝外卖团队可能需要维护两个版本，一个自身版本，另一个是淘宝客户端版本，这样维护成本和沟通成本会比较高。况且淘宝外卖不只是接入淘宝客户端，它还可以接入到其他应用中，比如支付宝应用，那么淘宝外卖团队维护的就不仅仅是两个版本了。
- 比如淘宝客户端接入了很多其他的应用，势必会使应用的体积急剧变大，编译时间会变得非常长，一个 Bug 和功能就会由组内的开发协作变为了组和组之间甚至是部门间的开发协作，极大地增加了开发测试成本和沟通成本，新功能的添加牵扯得越多，版本发布的时间变得越不可控。

3. 65536 限制，内存占用大

在 13.4.1 节中我们知道了 65536 限制，随着应用的代码量不断增大，引入的库也越来越多，特别是应用需要接入其他应用，那么方法数很容易超过 65536 个。应用代码量的增加同时也导致了应用占用大量的内存。

15.2.2 插件化思想

Android 系统本身并没有提供太多的功能，内置的应用数量和整体功能也很有限，它像是一个为人类服务的机器人，只能满足人类基本的需求，如图 15-2 所示。

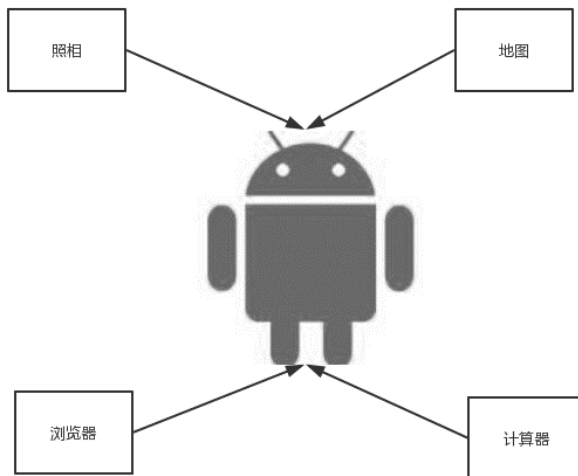


图 15-2 初始的 Android 系统

可以看到初始的机器人只有照相、地图、浏览器、计算机等功能，这显然是有些乏味的，我们可以给这个机器人安装很多其他的应用，使它提供更多的功能，如图 15-3 所示。

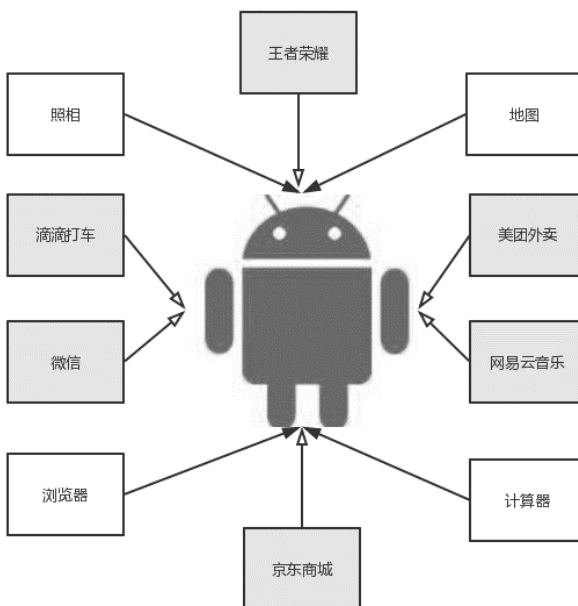


图 15-3 安装应用后的 Android 系统

我们给这个机器人安装了很多应用，这些应用不仅覆盖了人的衣食住行，还提供了娱乐功能，我们可以玩游戏、听音乐和购物等，机器人的功能也得到了极大提升，能够为人

类提供更多的服务。这些安装的应用可以理解为插件，这些插件可以自由地进行插拔，比如我们需要玩游戏时可以安装“王者荣耀”，如果不好玩就把它卸载掉。这么说来其实 Android、iOS、Mac 等操作系统采用的都是这种思想，也就是插件化思想。

15.2.3 插件化定义

15.2.1 节所提出的问题就可以用插件化的思想来解决，如果没有采用插件化，那么手机淘宝客户端的框架可以缩略地理解为如图 15-4 所示的样子。

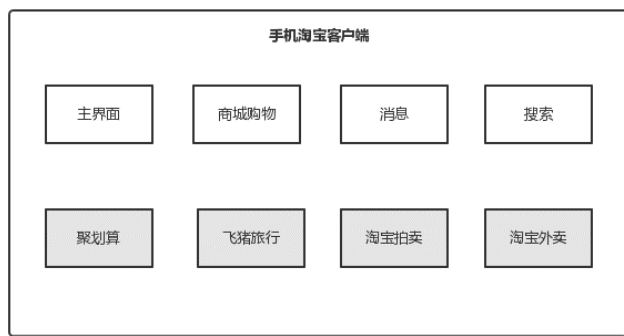


图 15-4 常规的缩略淘宝主客户端

图 15-4 中采用常规技术的淘宝客户端会分为两大部分，一个是自身的业务模块，比如商城购物、消息和搜索等，另一个是要外接的其他的应用业务，比如聚划算、飞猪旅行和淘宝外卖等。如果采用这种常规的技术方案，那么会产生 15.2.1 节中提出的各种问题，为了解决这些问题，我们可以采用插件化思想来对淘宝主客户端框架进行改造，如图 15-5 所示。

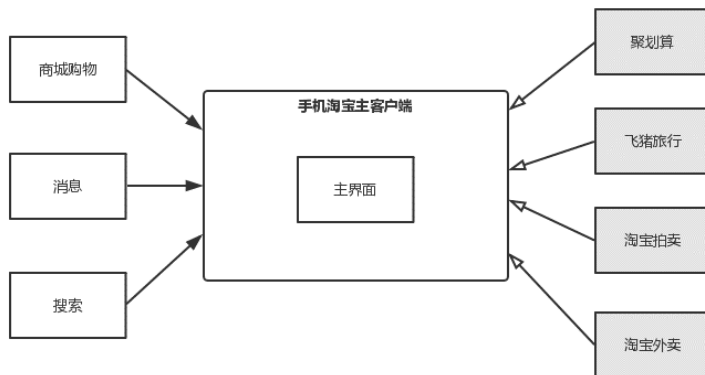


图 15-5 采用插件化思想的淘宝主客户端

插件化的客户端由宿主和插件两个部分组成，宿主就是指先被安装到手机中的 APK，就是平常我们加载的普通 APK。插件一般是指经过处理的 APK、so 和 dex 等文件，插件可以被宿主进行加载，有的插件也可以作为 APK 独立运行。可以看出采用插件化的淘宝客户端分为了两大部分，一部分是宿主部分，也就是淘宝主客户端，其内部包含了主界面模块；另一部分是插件部分，不仅包括了外接的其他应用业务，比如聚划算和飞猪旅行，同时也包括了淘宝自身的业务模块，比如消息和搜索。需要注意的是，这里的举例更多是为了便于理解，只是淘宝客户端演进过程中的一个非常缩略的框架，和真实的淘宝客户端有非常大的区别。讲到这里就可以引出插件化的定义：将一个应用按照插件的方式进行改造的过程就叫作插件化。采用了插件化的淘宝主客户端就避免了 15.2.1 节提出的各种问题，在协作方面，插件可以由一个人或者一个小组来进行开发，这样各个插件之间，以及插件和宿主之间的耦合度会降低。应用间的接入和维护也变得便捷，每个应用团队只需要负责自己的那一部分就可以了。应用以及主 dex 的体积也会相应变小，间接地避免了 65536 限制。第一次加载到内存的只有淘宝主客户端，当使用到其他插件时才会加载相应插件到内存，这样就减少了内存的占用。

15.3 插件化框架对比

为了方便地将应用插件化，出现了很多插件化框架。虽然插件化框架最近两年才受到广泛关注，但其实在 2012 年大众点评的屠毅敏就推出了 AndroidDynamicLoader 框架，这是最早的插件化框架。插件化技术发展到现在，已经产生了众多插件化框架，如表 15-1 所示。

表 15-1 插件化框架

插件化框架	作 者	插件化框架	作 者
DynamicAPK	携程	dynamic-load-apk	任玉刚
DroidPlugin	360	Small	Wequick
RePlugin	360	VirtualApk	滴滴

目前主流的插件化方案对比如表 15-2 所示。

表 15-2 主流的插件化方案对比

特 性	VirtualApk	DroidPlugin	Small	RePlugin
支持四大组件	全支持	全支持	只支持 Activity	全支持
组件无须在宿主 manifest 中预注册	Yes	Yes	Yes	Yes

续表

特 性	VirtualApk	DroidPlugin	Small	RePlugin
插件可以依赖宿主	Yes	No	Yes	Yes
支持 PendingIntent	Yes	Yes	No	Yes
Android 特性支持	几乎全部	几乎全部	大部分	几乎全部
兼容性适配	高	高	中等	高
插件构建	Gradle 插件	无	Gradle 插件	Gradle 插件

如果加载的插件不需要和宿主有任何耦合，也无须和宿主进行通信，比如加载第三方 App，那么推荐使用 RePlugin，其他的情况推荐使用 VirtualApk。由于 VirtualApk 在加载耦合插件方面是插件化框架的首选，具有普遍的适用性，本章会结合 VirtualApk 来讲解插件化的原理。首先编写简单的例子实现 Activity、Service 插件化，起一个知识储备和过度的作用，然后在广播、ContentProvider、资源和 so 的插件化中讲解 VirtualApk 是如何实现的，这样更有助于理解插件化的原理。

15.4 Activity 插件化

四大组件的插件化是插件化技术的核心知识点，而 Activity 插件化更是重中之重，Activity 插件化主要有 3 种实现方式，分别是反射实现、接口实现和 Hook 技术实现。反射实现会对性能有所影响，主流的插件化框架没有采用此方式，关于接口实现可以阅读 dynamic-load-apk 的源码，这里不做介绍，目前 Hook 技术实现是主流，因此本章主要介绍 Hook 技术实现。

Hook 技术实现主要有两种解决方案，一种是通过 Hook IActivityManager 来实现，另一种是 Hook Instrumentation 实现。在讲到这两个解决方案前，我们需要从整体上了解 Activity 的启动流程。

15.4.1 Activity 的启动过程回顾

Activity 的启动过程主要分为两种，一种是根 Activity 的启动过程，另一种是普通 Activity 的启动过程。关于根 Activity 的启动过程在 4.1 节介绍过，这里简单回顾一下，如图 15-6 所示。

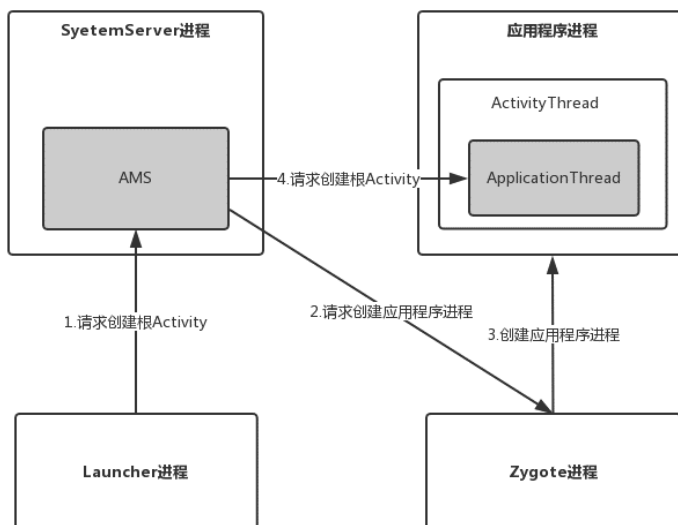


图 15-6 根 Activity 启动过程

首先 Launcher 进程向 AMS 请求创建根 Activity，AMS 会判断根 Activity 所需的应用程序进程是否存在并启动，如果不存在就会请求 Zygote 进程创建应用程序进程。应用程序进程启动后，AMS 会请求应用程序进程创建并启动根 Activity。普通 Activity 和根 Activity 的启动过程大同小异，但是没有这么复杂，因为不涉及应用程序进程的创建，与 Launcher 也没关系，如图 15-7 所示。

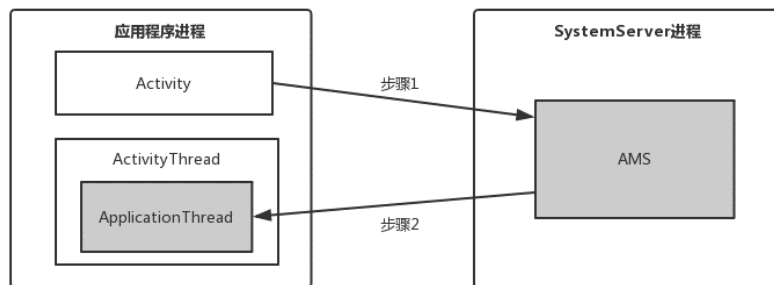


图 15-7 普通 Activity 启动过程

图 15-7 抽象地给出了普通 Activity 的启动过程。在应用程序进程中的 Activity 向 AMS 请求创建普通 Activity（步骤 1），AMS 会对这个 Activity 的生命周期和栈进行管理，校验 Activity 等，关于栈管理请查看 6.5 节。如果 Activity 满足 AMS 的校验，AMS 就会请求应用程序进程中的 ActivityThread 去创建并启动普通 Activity（步骤 2）。

15.4.2 Hook IActivityManager方案实现

AMS 存在于 SystemServer 进程中, 我们无法直接修改, 只能在应用程序进程中做文章。可以采用预先占坑的方式来解决没有在 AndroidManifest.xml 中显式声明的问题, 具体做法就是在图 15-7 所示的步骤 1 之前使用一个在 AndroidManifest.xml 中注册的 Activity 来进行占坑, 用来通过 AMS 的校验。接着在步骤 2 之后用插件 Activity 替换占坑的 Activity, 接下来根据这个解决方案我们来实践一下。

15.4.2.1 注册 Activity 进行占坑

为了更好地讲解启动插件 Activity 的原理, 这里省略了插件 Activity 的加载逻辑, 直接创建一个 TargetActivity 来代表已经加载进来的插件 Activity, 接着我们再创建一个 SubActivity 用来占坑。在 AndroidManifest.xml 中注册 SubActivity, 如下所示:

AndroidManifest.xml

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.example.liuwangshu.pluginactivity">S
    <application
        ...
        <activity android:name=".StubActivity"/>
    </application>
</manifest>
```

TargetActivity 用来代表已经加载进来的插件 Activity, 因此不需要在 AndroidManifest.xml 中进行注册。如果我们直接在 MainActivity 中启动 TargetActivity 肯定会报错 (android.content.ActivityNotFoundException 异常)。

15.4.2.2 使用占坑 Activity 通过 AMS 验证

为了防止报错, 需要将启动的 TargetActivity 替换为 SubActivity, 用 SubActivity 来通过 AMS 的验证。在第 6 章中讲过 Android 8.0 与 Android 7.0 的 AMS 家族有一些差别, 主要是 Android 8.0 去掉了 AMS 的代理 ActivityManagerProxy, 代替它的是 IActivityManager, 直接采用 AIDL 来进行进程间通信。Android 7.0 的 Activity 的启动会调用 ActivityManagerNative 的 getDefault 方法, 如下所示:

frameworks/base/core/java/android/app/ActivityManagerNative.java

```
static public IActivityManager getDefault() {
    return gDefault.get();
}
```



```
private static final Singleton<IActivityManager> gDefault = new Singleton
<IActivityManager>() {
    protected IActivityManager create() {
        IBinder b = ServiceManager.getService("activity");
        if (false) {
            Log.v("ActivityManager", "default service binder = " + b);
        }
        IActivityManager am = asInterface(b);
        if (false) {
            Log.v("ActivityManager", "default service = " + am);
        }
        return am;
    }
};
```

getDefault 方法返回了 IActivityManager 类型的对象, IActivityManager 借助了 Singleton 类来实现单例, 而且 gDefault 又是静态的, 因此 IActivityManager 是一个比较好的 Hook 点。Android 8.0 的 Activity 的启动会调用 ActivityManager 的 getService 方法, 如下所示:

frameworks/base/core/java/android/app/ActivityManager.java

```
public static IActivityManager getService() {
    return IActivityManagerSingleton.get();
}

private static final Singleton<IActivityManager> IActivityManagerSingleton =
    new Singleton<IActivityManager>() {
        @Override
        protected IActivityManager create() {
            final IBinder b = ServiceManager.getService(Context.ACTIVITY_
                SERVICE);
            final IActivityManager am = IActivityManager.Stub.asInterface(b);
            return am;
        }
    };
```

同样地, getService 方法返回了 IActivityManager 类型的对象, 并且 IActivityManager 借助了 Singleton 类来实现单例, 再结合图 6-1 和 6-3, 确定了无论是 Android 7.0 还是 Android 8.0, IActivityManager 都是比较好的 Hook 点。Singleton 类如下所示, 后面会用到:

frameworks/base/core/java/android/util/Singleton.java

```
public abstract class Singleton<T> {
    private T mInstance;
    protected abstract T create();
}
```

```

    public final T get() {
        synchronized (this) {
            if (mInstance == null) {
                mInstance = create();
            }
            return mInstance;
        }
    }
}

```

由于 Hook 需要多次对字段进行反射操作，先写一个字段工具类 FieldUtil：

FieldUtil.java

```

public class FieldUtil {
    public static Object getField(Class clazz, Object target, String name) throws
    Exception {
        Field field = clazz.getDeclaredField(name);
        field.setAccessible(true);
        return field.get(target);
    }
    public static Field getField(Class clazz, String name) throws Exception{
        Field field = clazz.getDeclaredField(name);
        field.setAccessible(true);
        return field;
    }
    public static void setField(Class clazz, Object target, String name, Object value)
    throws Exception {
        Field field = clazz.getDeclaredField(name);
        field.setAccessible(true);
        field.set(target, value);
    }
}

```

其中 setField 方法不会马上用到，接着定义替换 IActivityManager 的代理类 IActivityManagerProxy，如下所示：

```

public class IActivityManagerProxy implements InvocationHandler {
    private Object mActivityManager;
    private static final String TAG = "IActivityManagerProxy";
    public IActivityManagerProxy(Object activityManager) {
        this.mActivityManager = activityManager;
    }
    @Override
    public Object invoke(Object o, Method method, Object[] args) throws Throwable {
        if ("startActivity".equals(method.getName())) { //1
            Intent intent = null;

```

```

        int index = 0;
        for (int i = 0; i < args.length; i++) {
            if (args[i] instanceof Intent) {
                index = i;
                break;
            }
        }
        intent = (Intent) args[index];
        Intent subIntent = new Intent();//2
        String packageName = "com.example.liuwangshu.pluginactivity";
        subIntent.setClassName(packageName,packageName+".StubActivity");//3
        subIntent.putExtra(HookHelper.TARGET_INTENT, intent);//4
        args[index] = subIntent;//5
    }
    return method.invoke(mActivityManager, args);
}
}

```

Hook 点 `IActivityManager` 是一个接口,建议采用动态代理。在注释 1 处拦截 `startActivity` 方法,接着获取参数 `args` 中第一个 `Intent` 对象,它原本要启动插件 `TargetActivity` 的 `Intent`。在注释 2、注释 3 处新建一个 `subIntent` 用来启动 `StubActivity`,在注释 4 处将这个 `TargetActivity` 的 `Intent` 保存到 `subIntent` 中,便于以后还原 `TargetActivity`。在注释 5 处用 `subIntent` 赋值给参数 `args`,这样启动的目标就变为了 `StubActivity`,用来通过 AMS 的校验。最后用代理类 `IActivityManagerProxy` 来替换 `IActivityManager`,如下所示:

HookHelper.java

```

public class HookHelper {
    public static final String TARGET_INTENT = "target_intent";
    public static void hookAMS() throws Exception {
        Object defaultSingleton=null;
        if (Build.VERSION.SDK_INT >= 26) {//1
            Class<?> activityManageClazz = Class.forName("android.app.ActivityManager");
            //获取 activityManager 中的 IActivityManagerSingleton 字段
            defaultSingleton= FieldUtil.getField(activityManageClazz ,null,
                "IActivityManagerSingleton");
        } else {
            Class<?> activityManagerNativeClazz = Class.forName("android.app.
                ActivityManagerNative");
            //获取 ActivityManagerNative 中的 gDefault 字段
            defaultSingleton= FieldUtil.getField(activityManagerNativeClazz,null,
                "gDefault");
        }
        Class<?> singletonClazz = Class.forName("android.util.Singleton");
    }
}

```

```

        Field mInstanceField= FieldUtil.getField(singletonClazz ,"mInstance");//2
        //获取 iActivityManager
        Object iActivityManager = mInstanceField.get(defaultSingleton);//3
        Class<?> iActivityManagerClazz = Class.forName("android.app.IActivityManager");
        Object proxy = Proxy.newProxyInstance(Thread.currentThread().getContext
        ClassLoader(),new Class<?>[] { iActivityManagerClazz }, new
        IActivityManagerProxy (iActivityManager));
        mInstanceField.set(defaultSingleton, proxy);
    }
}

```

首先在注释 1 处对系统版本进行区分，最终获取的是 Singleton<IActivityManager>类型的 IActivityManagerSingleton 或者 gDefault 字段。在注释 2 处获取 Singleton 类中的 mInstance 字段，从前面 Singleton 类的代码可以得知 mInstance 字段的类型为 T，在注释 3 处得到 IActivityManagerSingleton 或者 gDefault 字段中的 T 的类型，T 的类型为 IActivityManager。最后动态创建代理类 IActivityManagerProxy，用 IActivityManagerProxy 来替换 IActivityManager。自定义一个 Application，在其中调用 HookHelper 的 hookAMS 方法，如下所示：

MyApplication.java

```

public class MyApplication extends Application {
    @Override
    public void attachBaseContext(Context base) {
        super.attachBaseContext(base);
        try {
            HookHelper.hookAMS();
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}

```

在 MainActivity 中启动 TargetActivity，如下所示：

MainActivity.java

```

public class MainActivity extends Activity {
    private Button bt_hook;
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        bt_hook = (Button) this.findViewById(R.id.bt_hook);
        bt_hook.setOnClickListener(new View.OnClickListener() {

```

```

@Override
public void onClick(View view) {
    Intent intent = new Intent(MainActivity.this, TargetActivity.class);
    startActivity(intent);
}
});
}
}

```

点击 Button 时,启动的并不是 TargetActivity 而是 SubActivity,同时 Log 中打印了“hook 成功”,说明我们已经成功用 SubActivity 通过了 AMS 的校验。

15.4.2.3 还原插件 Activity

前面用占坑 Activity 通过了 AMS 的校验,但是我们要启动的是插件 TargetActivity,还需要用插件 TargetActivity 来替换占坑的 SubActivity,这一替换的时机就在图 15-7 所示的步骤 2 之后。在 4.1.3 节中讲到了 ActivityThread 启动 Activity 的过程,如图 15-8 所示。

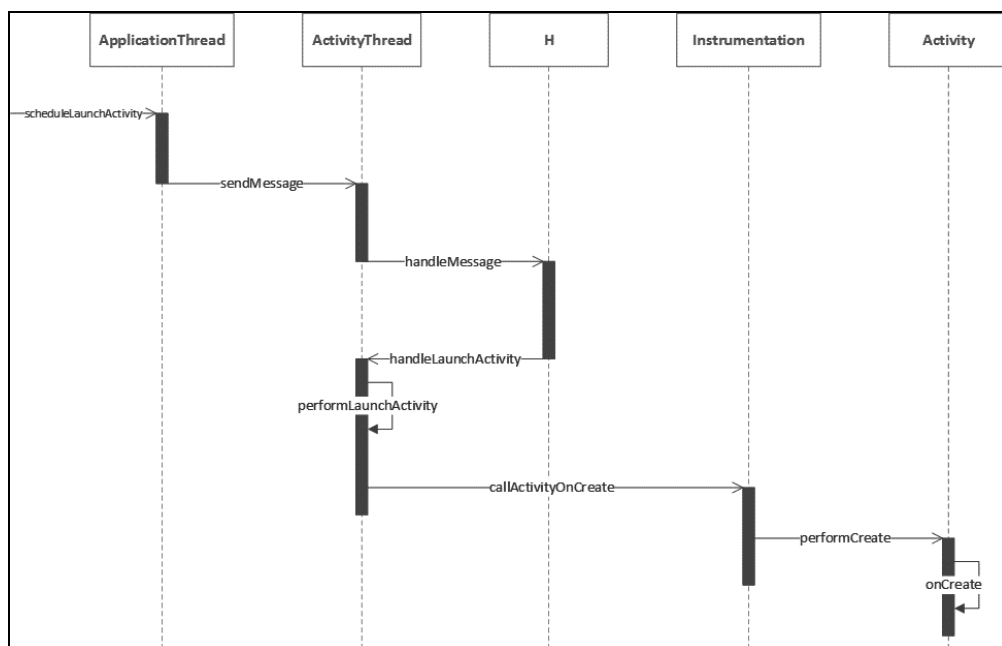


图 15-8 ActivityThread 启动 Activity 的过程

ActivityThread 会通过 H 类将代码的逻辑切换到主线程中, H 类是 ActivityThread 的内部类并继承自 Handler, 如下所示:

```
frameworks/base/core/java/android/app/ActivityThread.java
```

```
private class H extends Handler {
    public static final int LAUNCH_ACTIVITY = 100;
    public static final int PAUSE_ACTIVITY = 101;
    ...
    public void handleMessage(Message msg) {
        if (DEBUG_MESSAGES) Slog.v(TAG, ">>> handling: " + codeToString(msg.
            what));
        switch (msg.what) {
            case LAUNCH_ACTIVITY: {
                Trace.traceBegin(Trace.TRACE_TAG_ACTIVITY_MANAGER, "activityStart");
                final ActivityClientRecord r = (ActivityClientRecord) msg.obj;

                r.packageInfo = getPackageInfoNoCheck(
                    r.activityInfo.applicationInfo, r.compatInfo);
                handleLaunchActivity(r, null, "LAUNCH_ACTIVITY");
                Trace.traceEnd(Trace.TRACE_TAG_ACTIVITY_MANAGER);
            } break;
            ...
        }
    }
    ...
}
```

H 类中重写的 handleMessage 方法会对 LAUNCH_ACTIVITY 类型的消息进行处理, 最终会调用 Activity 的 onCreate 方法。那么在哪进行替换呢? 接着来看 Handler 的 dispatchMessage 方法:

```
frameworks/base/core/java/android/os/Handler.java
```

```
public void dispatchMessage(Message msg) {
    if (msg.callback != null) {
        handleCallback(msg);
    } else {
        if (mCallback != null) {
            if (mCallback.handleMessage(msg)) {
                return;
            }
        }
        handleMessage(msg);
    }
}
```

Handler 的 dispatchMessage 用于处理消息, 看到如果 Handler 的 Callback 类型的 mCallback 不为 null, 就会执行 mCallback 的 handleMessage 方法。因此, mCallback 可以作

为 Hook 点, 我们可以用自定义的 Callback 来替换 mCallback, 自定义的 Callback 如下所示:

HCallback.java

```
public class HCallback implements Handler.Callback{
    public static final int LAUNCH_ACTIVITY = 100;
    Handler mHandler;
    public HCallback(Handler handler) {
        mHandler = handler;
    }
    @Override
    public boolean handleMessage(Message msg) {
        if (msg.what == LAUNCH_ACTIVITY) {
            Object r = msg.obj;
            try {
                //得到消息中的 Intent(启动 SubActivity 的 Intent)
                Intent intent = (Intent) FieldUtil.getField(r.getClass(), r, "intent");
                //得到此前保存起来的 Intent(启动 TargetActivity 的 Intent)
                Intent target = intent.getParcelableExtra(HookHelper.TARGET_INTENT);
                //将启动 SubActivity 的 Intent 替换为启动 TargetActivity 的 Intent
                intent.setComponent(target.getComponent());
            } catch (Exception e) {
                e.printStackTrace();
            }
        }
        mHandler.handleMessage(msg);
        return true;
    }
}
```

HCallback 实现了 Handler.Callback, 并重写了 handleMessage 方法, 当收到消息的类型为 LAUNCH_ACTIVITY 时, 将启动 SubActivity 的 Intent 替换为启动 TargetActivity 的 Intent。接着我们在 HookHelper 中定义一个 hookHandler 方法, 如下所示:

HookHelper.java

```
public static void hookHandler() throws Exception {
    Class<?> activityThreadClass = Class.forName("android.app.ActivityThread");
    Object currentActivityThread= FieldUtil.getField(activityThreadClass, null,
        "sCurrentActivityThread");//1
    Field mHField = FieldUtil.getField(activityThread, "mH");//2
    Handler mH = (Handler) mHField.get(currentActivityThread);//3
    FieldUtil.setField(Handler.class, mH, "mCallback", new HCallback(mH));
}
```

ActivityThread 类中有一个静态变量 sCurrentActivityThread，用于表示当前的 ActivityThread 对象，因此在注释 1 处获取 ActivityThread 中定义的 sCurrentActivityThread 对象。注释 2 处获取 ActivityThread 类的 mH 字段，接着在注释 3 处获取当前 ActivityThread 对象中的 mH 对象，最后用 HCallback 来替换 mH 中的 mCallback。在 MyApplication 的 attachBaseContext 方法中调用 HookHelper 的 hookHandler 方法，运行程序，当我们单击“启动插件”按钮时，发现启动的是插件 TargetActivity。

15.4.2.4 插件 Activity 的生命周期

插件 TargetActivity 确实启动了，但是它有生命周期吗？这里从源码角度来进行分析，Activity 的 finish 方法可以触发 Activity 的生命周期变化，和 Activity 的启动过程类似，finish 方法如下所示：

frameworks/base/core/java/android/app/Activity.java

```
public void finish() {
    finish(DONT_FINISH_TASK_WITH_ACTIVITY);
}
private void finish(int finishTask) {
    if (mParent == null) {
        int resultCode;
        Intent resultData;
        synchronized (this) {
            resultCode = mResultCode;
            resultData = mResultData;
        }
        if (false) Log.v(TAG, "Finishing self: token=" + mToken);
        try {
            if (resultData != null) {
                resultData.prepareToLeaveProcess(this);
            }
            if (ActivityManager.getService()
                .finishActivity(mToken, resultCode, resultData, finishTask))
                { //1
                    mFinished = true;
                }
        } catch (RemoteException e) {
            // Empty
        }
    } else {
        mParent.finishFromChild(this);
    }
}
```


finish 方法的调用链和 Activity 的启动过程是类似的，在注释 1 处调用 AMS 的 finishActivity 方法，接着是 AMS 通过 ApplicationThread 调用 ActivityThread，ActivityThread 向 H 类发送 DESTROY_ACTIVITY 类型的消息，H 类接收到这个消息会执行 handleDestroyActivity 方法，handleDestroyActivity 方法又调用了 performDestroyActivity 方法，如下所示：

```
frameworks/base/core/java/android/app/ActivityThread.java
```

```
private ActivityClientRecord performDestroyActivity(IBinder token, boolean
finishing, int configChanges, boolean getNonConfigInstance) {
    ActivityClientRecord r = mActivities.get(token); //1
    ...
    try {
        r.activity.mCalled = false;
        mInstrumentation.callActivityOnDestroy(r.activity); //2
        ...
    } catch (SuperNotCalledException e) {
        ...
    }
}
mActivities.remove(token);
StrictMode.decrementExpectedActivityCount(activityClass);
return r;
```

在注释 1 处通过 IBinder 类型的 token 来获取 ActivityClientRecord，ActivityClientRecord 用于描述应用进程中的 Activity。在注释 2 处调用 Instrumentation 的 callActivityOnDestroy 方法来调用 Activity 的 OnDestroy 方法，并传入了 r.activity。前面的例子我们用 SubActivity 替换了 TargetActivity 通过了 AMS 的校验，这样 AMS 只知道 SubActivity 的存在，那么 AMS 是如何能控制 TargetActivity 生命周期的回调的呢？我们接着往下看，启动 Activity 时会调用 ActivityThread 的 performLaunchActivity 方法，如下所示：

```
frameworks/base/core/java/android/app/ActivityThread.java
```

```
private Activity performLaunchActivity(ActivityClientRecord r, Intent customIntent) {
    ...
    java.lang.ClassLoader cl = appContext.getClassLoader();
    activity = mInstrumentation.newActivity(
        cl, component.getClassName(), r.intent); //1
    ...
    activity.attach(appContext, this, getInstrumentation(), r.token, r.ident,
    app, r.intent, r.activityInfo, title, r.parent, r.embeddedID,
    r.lastNonConfigurationInstances, config, r.referrer,
    r.voiceInteractor, window, r.configCallback);
```

```

        ...
        mActivities.put(r.token, r); //2
        ...
    return activity;
}

```

在注释 1 处根据 Activity 的类名用 ClassLoader 加载 Activity，接着调用 Activity 的 attach 方法，将 r.token 赋值给 Activity 的成员变量 mToken。在注释 2 处将 ActivityClientRecord 根据 r.token 保存在 mActivities 中 (mActivities 类型为 ArrayMap<IBinder, ActivityClientRecord>)，再结合 Activity 的 finish 方法的注释 1 处，可以得出结论：AMS 和 ActivityThread 之间的通信采用了 token 来对 Activity 进行标识，并且此后的 Activity 的生命周期处理也是根据 token 来对 Activity 进行标识的。回到这个例子来，我们在 Activity 启动时用插件 TargetActivity 替换占坑 SubActivity，这一过程在 performLaunchActivity 方法调用之前，因此注释 2 处的 r.token 指向的是 TargetActivity，在 performDestroyActivity 的注释 1 处获取的就是代表 TargetActivity 的 ActivityClientRecord，可见 TargetActivity 是具有生命周期的。

15.4.3 Hook Instrumentation 方案实现

Hook Instrumentation 实现要比 Hook IActivityManager 实现简单一些，示例代码会和 Hook IActivityManager 实现有重复，重复的部分这里不再赘述。Hook Instrumentation 实现同样也需要用到占坑 Activity，与 Hook IActivityManager 实现不同的是，用占坑 Activity 替换插件 Activity 以及还原插件 Activity 的地方不同。Activity 的 startActivity 方法调用时序图如图 15-9 所示。

从图 15-9 可以发现，在 Activity 通过 AMS 校验前，会调用 Activity 的 startActivityForResult 方法：

frameworks/base/core/java/android/app/Activity.java

```

public void startActivityForResult(@RequiresPermission Intent intent, int requestCode,
    @Nullable Bundle options) {
    if (mParent == null) {
        options = transferSpringboardActivityOptions(options);
        Instrumentation.ActivityResult ar =
            mInstrumentation.execStartActivity(
                this, mMainThread.getApplicationThread(), mToken, this,
                intent, requestCode, options);
        ...
    } else {

```

```

    ...
}
}

```

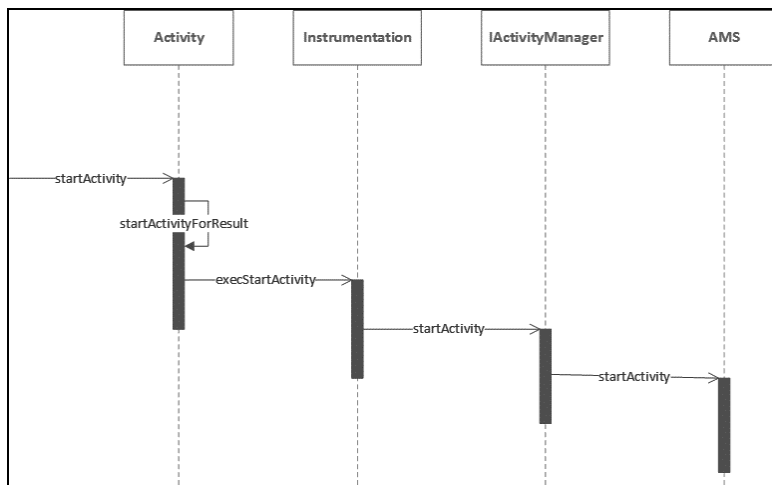


图 15-9 Activity 的 startActivity 方法调用时序图

在 startActivityForResult 方法中调用了 Instrumentation 的 execStartActivity 方法来激活 Activity 的生命周期。

图 15-8 中会调用 ActivityThread 的 performLaunchActivity 方法，如下所示：

frameworks/base/core/java/android/app/ActivityThread.java

```

private Activity performLaunchActivity(ActivityClientRecord r, Intent customIntent) {
    ...
    //创建要启动 Activity 的上下文环境
    ContextImpl appContext = createBaseContextForActivity(r);
    Activity activity = null;
    try {
        java.lang.ClassLoader cl = appContext.getClassLoader();
        //用类加载器来创建 Activity 的实例
        activity = mInstrumentation.newActivity(
            cl, component.getClassName(), r.intent); //1
        ...
    } catch (Exception e) {
        ...
    }
    ...
    return activity;
}

```

在注释 1 处调用了 `mInstrumentation` 的 `newActivity` 方法，其内部会用类加载器来创建 `Activity` 的实例。看到这里我们可以得到方案，就是在 `Instrumentation` 的 `execStartActivity` 方法中用占坑 `SubActivity` 来通过 `AMS` 的验证，在 `Instrumentation` 的 `newActivity` 方法中还原 `TargetActivity`，这两部操作都和 `Instrumentation` 有关，因此我们可以用自定义的 `Instrumentation` 来替换掉 `mInstrumentation`。首先我们自定义一个 `Instrumentation`，在 `execStartActivity` 方法中将启动的 `TargetActivity` 替换为 `SubActivity`，如下所示：

InstrumentationProxy.java

```
public class InstrumentationProxy extends Instrumentation {
    private Instrumentation mInstrumentation;
    private PackageManager mPackageManager;
    public InstrumentationProxy(Instrumentation instrumentation, PackageManager
    packageManager) {
        mInstrumentation = instrumentation;
        mPackageManager = packageManager;
    }
    public ActivityResult execStartActivity(
        Context who, IBinder contextThread, IBinder token, Activity target,
        Intent intent, int requestCode, Bundle options) {
        List<ResolveInfo> infos = mPackageManager.queryIntentActivities(intent,
        PackageManager.MATCH_ALL);
        if (infos == null || infos.size() == 0) {
            intent.putExtra(HookHelper.TARGET_INTENS_NAME,
            intent.getComponent().getClassName()); //1
            intent.setClassName(who,
            "com.example.liuwangshu.pluginactivity.StubActivity"); //2
        }
        try {
            Method execMethod = Instrumentation.class.getDeclaredMethod("execStart
            Activity", Context.class, IBinder.class, IBinder.class, Activity.class,
            Intent.class, int.class, Bundle.class);
            return (ActivityResult) execMethod.invoke(mInstrumentation, who,
            contextThread, token, target, intent, requestCode, options);
        } catch (Exception e) {
            e.printStackTrace();
        }
        return null;
    }
}
```

首先查找要启动的 `Activity` 是否已经在 `AndroidManifest.xml` 中注册了，如果没有注册就在注释 1 处将要启动的 `Activity(TargetActivity)` 的 `ClassName` 保存起来用于后面还原

TargetActivity，接着在注释 2 处替换要启动的 Activity 为 StubActivity，最后通过反射调用 execStartActivity 方法，这样就可以用 StubActivity 通过 AMS 的验证。在 InstrumentationProxy 的 newActivity 方法中还原 TargetActivity，如下所示：

InstrumentationProxy.java

```
public Activity newActivity(ClassLoader cl, String className, Intent intent) throws
InstantiationException,
    IllegalAccessException, ClassNotFoundException {
    String intentName = intent.getStringExtra(HookHelper.TARGET_INTENT_NAME);
    if (!TextUtils.isEmpty(intentName)) {
        return super.newActivity(cl, intentName, intent);
    }
    return super.newActivity(cl, className, intent);
}
```

在 newActivity 方法中创建了此前保存的 TargetActivity，完成了还原 TargetActivity。编写 hookInstrumentation 方法，用 InstrumentationProxy 替换 mInstrumentation：

HookHelper.java

```
public static void hookInstrumentation(Context context) throws Exception {
    Class<?> contextImplClass = Class.forName("android.app.ContextImpl");
    Field mMainThreadField = FieldUtil.getField(contextImplClass,
        "mMainThread");//1
    Object activityThread = mMainThreadField.get(context);//2
    Class<?> activityThreadClass = Class.forName("android.app.ActivityThread");
    Field mInstrumentationField=FieldUtil.getField(activityThreadClass,
        "mInstrumentation");//3
    FieldUtil.setField(activityThreadClass,activityThread,"mInstrumentation",
        new InstrumentationProxy((Instrumentation) mInstrumentationField.
            get(activityThread), context.getPackageManager()));
}
```

在注释 1 处获取 ContextImpl 类的 ActivityThread 类型的 mMainThread 字段，在注释 2 处获取当前上下文环境的 ActivityThread 对象。在注释 3 处获取 ActivityThread 类中的 mInstrumentation 字段，最后用 InstrumentationProxy 来替换 mInstrumentation。在 MyApplication 的 attachBaseContext 方法中调用 HookHelper 的 hookInstrumentation 方法，运行程序，当我们单击“启动插件”按钮时，发现启动的是插件 TargetActivity。

15.4.4 总结

这一节我们学习了启动插件 Activity 的原理，主要的方案就是先用一个在

AndroidManifest.xml 中注册的 Activity 来进行占坑，用来通过 AMS 的校验，接着在合适的时机用插件 Activity 替换占坑的 Activity。为了更好地讲解启动插件 Activity 的原理，本节省略了插件 Activity 的加载逻辑，直接创建一个 TargetActivity 来代表已经加载进来的插件 Activity。同时这一节使我们更好地理解 Activity 的启动过程。

15.5 Service 插件化

Service 插件化和 Activity 插件化的原理有些不同，我们先来回顾一下 Service 的启动过程。

15.5.1 插件化方面 Service 与 Activity 的不同

在 4.2 节讲到了 Service 的启动过程，它和 Activity 的启动过程类似，但有些不同，ContextImpl 到 AMS 的调用过程如图 15-10 所示。

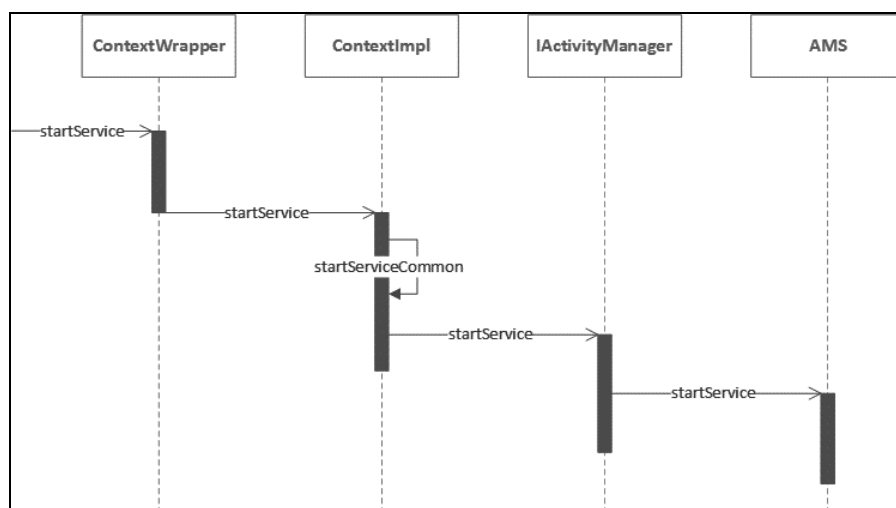


图 15-10 ContextImpl 到 AMS 的调用过程

在 ContextImpl 到 AMS 的调用过程中并没有交由 Instrumentation 来处理，在后续的 ActivityThread 启动 Service 过程中也是一样的，如图 15-11 所示。

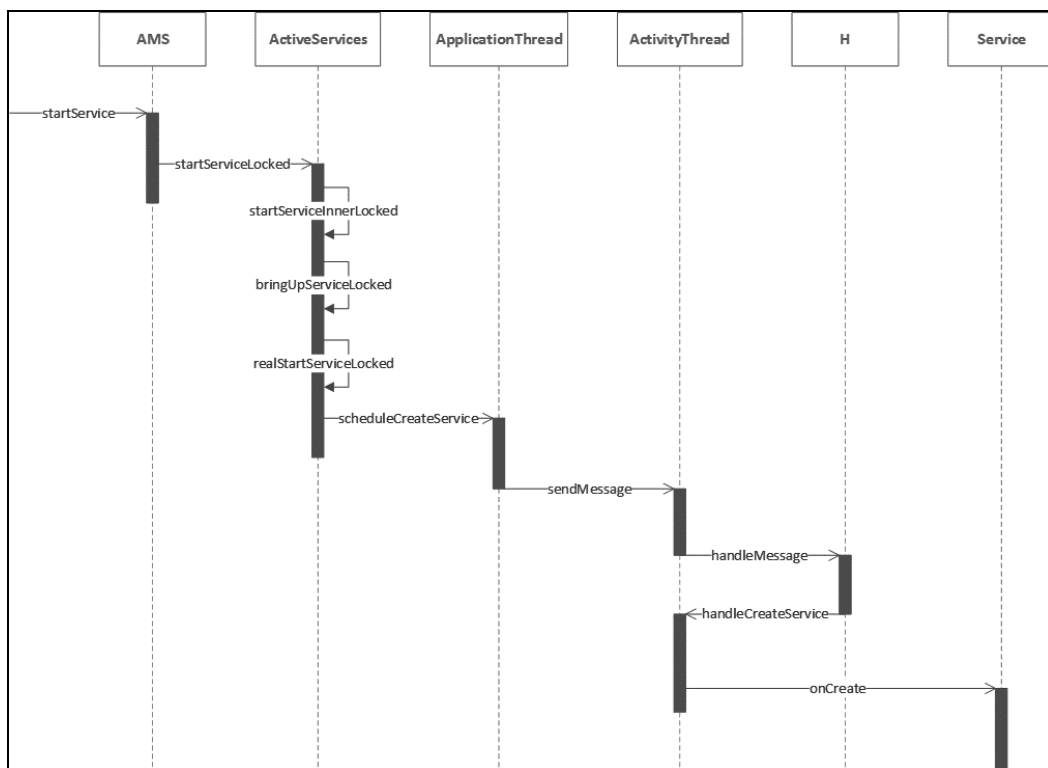


图 15-11 ActivityThread 启动 Service

可见 Service 的启动和 Instrumentation 完全没有关联,因此 Service 插件化不能通过 Hook Instrumentation 来实现。Service 插件化可以用 Hook IActivityManager 的方案来实现吗? 带着这个疑问,我们需要了解在插件化方面 Activity 和 Service 有何不同,主要有以下 3 点:

- Activity 是基于栈管理的,一个栈中的 Activity 的数量不会太多,因此插件化框架处理的插件 Activity 数量是有限的,可以声明有限的占坑 Activity 来实现。除去硬件和系统限制,插件化框架处理的插件 Service 的数量可以是近乎无限的,无法用有限的占坑 Service 来实现。
- 在 Standard 模式下多次启动同一个占坑 Activity 可以创建多个 Activity 实例,但是多次启动占坑 Service 并不会创建多个 Service 实例。
- 用户和界面的交互会影响到 Activity 的生命周期,因此插件 Activity 的生命周期需要交由系统来管理,Hook IActivityManager 方案中还原插件 Activity 就是为了这一点。Service 的生命周期不受用户影响,可以由开发者管理生命周期,没有必要还原插件。

综合上面 3 点得出的结论就是，Service 插件化不可以用 Hook IActivityManager 方案来实现，我们需要找到一个新的方案。

15.5.2 代理分发实现

Activity 插件化的重点在于要保证它的生命周期，而 Service 插件化的重点是保证它的优先级，这就需要用个真正的 Service 来实现，而不是像占坑 Activity 那样起一个占坑的作用。当启动插件 Service 时，就会先启动代理 Service，当这个代理 Service 运行起来之后，在它的 onStartCommand 等方法里面进行分发，执行插件 TargetService 的 onCreate 等方法，这一方案就叫作代理分发。

15.5.2.1 启动代理 Service

Service 插件化需要一个真正的 Service 来实现，我们先要在 AndroidManifest.xml 中注册代理 ProxyService，如下所示：

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.example.liuwangshu.pluginbservice">

    <application
        ...
        <service android:name=".ProxyService"
            //android:process=":daemon"
        />
    </application>
</manifest>
```

这里我们只是注册了 ProxyService，让它运行在当前应用程序进程中，你也可以定义 android:process 用于支持多进程。在 MainActivity 中启动插件 Service，如下所示：

MainActivity.java

```
public class MainActivity extends AppCompatActivity {
    private Button bt_hook;
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        bt_hook = (Button) this.findViewById(R.id.bt_hook);
        bt_hook.setOnClickListener(new View.OnClickListener() {
            @Override
```



```

        public void onClick(View view) {
            Intent intent = new Intent(MainActivity.this, TargetService.class);
            startService(intent);
        }
    });
}
}

```

为了便于理解, 这里省略了插件 Service 的加载逻辑, 直接创建一个 TargetService 来代表已经加载进来的插件 Service。TargetService 的代码如下所示:

TargetService.java

```

public class TargetService extends Service {
    private static final String TAG = "TargetService";
    @Nullable
    @Override
    public IBinder onBind(Intent intent) {
        return null;
    }
    @Override
    public void onCreate() {
        super.onCreate();
        Log.d(TAG, "onCreate");
    }
    @Override
    public int onStartCommand(Intent intent, int flags, int startId) {
        Log.d(TAG, "onStartCommand");
        return super.onStartCommand(intent, flags, startId);
    }
}

```

TargetService 只是用来打印 Log, 确定自身的方法是否已经被调用。TargetService 用来代表插件 Service, 不能够直接启动, 需要先启动代理 ProxyService, 为了达到这一目的我们需要 Hook IActivityManager, 具体的原理和步骤与 15.4.2 节讲得类似, 定义替换 IActivityManager 的代理类 IActivityManagerProxy, 如下所示:

```

public class IActivityManagerProxy implements InvocationHandler {
    private Object mActivityManager;
    private static final String TAG = "IActivityManagerProxy";
    public IActivityManagerProxy(Object activityManager) {
        this.mActivityManager = activityManager;
    }
    @Override
    public Object invoke(Object o, Method method, Object[] args) throws Throwable {

```

```

        if ("startService".equals(method.getName())) { //1
            Intent intent = null;
            int index = 0;
            for (int i = 0; i < args.length; i++) {
                if (args[i] instanceof Intent) {
                    index = i;
                    break;
                }
            }
            intent = (Intent) args[index];
            Intent proxyIntent = new Intent(); //2
            String packageName = "com.example.liuwangshu.plugin.service";
            proxyIntent.setClassName(packageName, packageName + ".ProxyService"); //3
            proxyIntent.putExtra(ProxyService.TARGET_SERVICE, intent.getComponent().
                getClassName()); //4
            args[index] = proxyIntent; //5
            Log.d(TAG, "Hook 成功");
        }
        return method.invoke(mActivityManager, args);
    }
}

```

在注释 1 处拦截 startService 方法，接着获取参数 args 中第一个 Intent 对象，它原本要启动插件 TargetActivity 的 Intent。在注释 2、注释 3 处新建一个 proxyIntent 用来启动 ProxyService，注释 4 处将这个 ProxyService 的 Intent 保存到 proxyIntent 中，便于此后进行分发。在注释 5 处将 proxyIntent 赋值给参数 args，这样启动的目标就变为了 ProxyService。定义了 IActivityManagerProxy，我们就需要用 IActivityManagerProxy 来替换系统的 IActivityManager，如下所示：

HookHelper.java

```

public class HookHelper {
    public static final String TARGET_INTENT = "target_intent";
    public static void hookAMS() throws Exception {
        Object defaultSingleton = null;
        if (Build.VERSION.SDK_INT >= 26) { //1
            Class<?> activityManageClazz = Class.forName("android.app.ActivityManager");
            //获取 activityManager 中的 IActivityManagerSingleton 字段
            defaultSingleton = FieldUtil.getField(activityManageClazz, null,
                "IActivityManagerSingleton");
        } else {
            Class<?> activityManagerNativeClazz = Class.forName("android.app.
                ActivityManagerNative");

```

```

        //获取 ActivityManagerNative 中的 gDefault 字段
        defaultSingleton= FieldUtil.getField(activityManagerNativeClazz,null,
        "gDefault");
    }
    Class<?> singletonClazz = Class.forName("android.util.Singleton");
    Field mInstanceField= FieldUtil.getField(singletonClazz ,"mInstance");//2
    //获取 iActivityManager
    Object iActivityManager = mInstanceField.get(defaultSingleton);//3
    Class<?> iActivityManagerClazz = Class.forName("android.app.IActivityManager");
    Object proxy = Proxy.newProxyInstance(Thread.currentThread().getContext
    ClassLoader(),new Class<?>[] { iActivityManagerClazz },
    new IActivityManagerProxy (iActivityManager));
    mInstanceField.set(defaultSingleton, proxy);
    }
}

```

首先在注释 1 处对系统版本进行区分,最终获取的是 Singleton<IActivityManager>类型的 IActivityManagerSingleton 或者 gDefault 字段。在注释 2 处获取 Singleton 类中的 mInstance 字段,从 15.4.2.2 节中给出的 Singleton 类代码可以得知 mInstance 字段的类型为 T,在注释 3 处得到 IActivityManagerSingleton 或者 gDefault 字段中的 T 的类型, T 的类型为 IActivityManager。最后动态创建代理类 IActivityManagerProxy,用 IActivityManagerProxy 来替换 IActivityManager。其中用到了 FieldUtil 类,这个类的定义在 14.5.1.2 节中已经讲过。自定义一个 Application 调用 HookHelper 的 hookAMS 方法,如下所示:

MyApplication.java

```

public class MyApplication extends Application {
    @Override
    protected void attachBaseContext(Context base) {
        super.attachBaseContext(base);
        try {
            HookHelper.hookAMS();
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}

```

运行应用单击 MainActivity 的按钮时,启动的不是插件 TargetService,而是代理 ProxyService,接下来我们需要在 ProxyService 中进行代理分发。

15.5.2.2 代理分发

编写 ProxyService 类，如下所示：

ProxyService.java

```
public class ProxyService extends Service {
    public static final String TARGET_SERVICE = "target_service";
    ...
    @Override
    public int onStartCommand(Intent intent, int flags, int startId) {
        if (null == intent || !intent.hasExtra(TARGET_SERVICE)) {
            return START_STICKY;
        }
        String serviceName = intent.getStringExtra(TARGET_SERVICE);
        if (null == serviceName) {
            return START_STICKY;
        }
        Service targetService=null;
        try {
            Class activityThreadClazz = Class.forName("android.app.ActivityThread");
            Method getActivityThreadMethod = activityThreadClazz.getDeclaredMethod(
                "getApplicationThread");
            getActivityThreadMethod.setAccessible(true);
            Object activityThread = FieldUtil.getField(activityThreadClazz, null,
                "sCurrentActivityThread");//1
            Object applicationThread = getActivityThreadMethod.invoke(activityThread);//2
            Class iInterfaceClazz = Class.forName("android.os.IInterface");
            Method asBinderMethod = iInterfaceClazz.getDeclaredMethod("asBinder");
            asBinderMethod.setAccessible(true);
            Object token = asBinderMethod.invoke(applicationThread);//3
            Class serviceClazz = Class.forName("android.app.Service");
            Method attachMethod = serviceClazz.getDeclaredMethod("attach",
                Context.class, activityThreadClazz, String.class, IBinder.class,
                Application.class, Object.class);
            attachMethod.setAccessible(true);
            Object defaultSingleton=null;
            if (Build.VERSION.SDK_INT >= 26) {//4
                Class<?> activityManageClazz = Class.forName("android.app.Activity
                    Manager");
                //获取 activityManager 中的 IActivityManagerSingleton 字段
                defaultSingleton = FieldUtil.getField(activityManageClazz, null,
                    "IActivityManagerSingleton");
            } else {
                Class<?> activityManagerNativeClazz = Class.forName("android.app.
```

```

        ActivityManagerNative");
        //获取 ActivityManagerNative 中的 gDefault 字段
        defaultSingleton = FieldUtil.getField(activityManagerNativeClazz,
            null, "gDefault");
    }
    Class<?> singletonClazz = Class.forName("android.util.Singleton");
    Field mInstanceField = FieldUtil.getField(singletonClazz, "mInstance");
    //获取 iActivityManager
    Object iActivityManager = mInstanceField.get(defaultSingleton);//5
    targetService = (Service) Class.forName(serviceName).newInstance();//6
    attachMethod.invoke(targetService, this, activityThread, intent.
        getComponent().getClassName(), token, getApplication(),
        iActivityManager);//7
    targetService.onCreate();
} catch (Exception e) {
    e.printStackTrace();
    return START_STICKY;
}
targetService.onStartCommand(intent, flags, startId);
return START_STICKY;
}
}

```

在 onStartCommand 方法中进行代理分发，这段代码比较长，主要做了 3 件事：

- ProxyService 需要长时间对 Service 进行分发处理，所以在参数条件不满足、出现异常和代码执行完毕时需要返回 START_STICKY，这样 ProxyService 会重新被创建并执行 onStartCommand 方法。
- 创建 targetService 并反射调用 targetService 的 attach 方法。
- 进行代理分发，执行 targetService 的 onCreate 方法。

这三件事中第二件事的代码比较多，简单来分析一下。为了反射调用 Service 的 attach 方法，除了要反射得到 attach 方法外，还需要得到 attach 方法需要的参数：ActivityThread、IBinder、IActivityManager 等。在注释 1 处得到 ActivityThread 对象，在注释 2 处根据 ActivityThread 得到 applicationThread 对象，得到 applicationThread 对象为的是在注释 3 处反射调用 ApplicationThread 的 asBinder 方法得到 token 对象，在源码中这个 token 是 IBinder 类型的。在注释 4 处到注释 5 处之间的代码逻辑在本章多次出现，我们应该非常熟悉了，是为了获取 iActivityManager。在注释 6 处反射得到 targetService，这里只是为了方便举例，用 targetService 代表已经加载进来的插件 Service，真正的插件化框架会用 ClassLoader 来加载插件中的 Service。在注释 7 处反射执行 targetService 的 attach 方法，并传入此前得到的

参数。最后执行 `targetService` 的 `onCreate` 方法来完成代理分发。运行应用单击 `MainActivity` 的按钮时，不仅启动了 `ProxyService`，插件 `TargetService` 也被启动了。

15.6 ContentProvider插件化

与 `Activity`、`BroadcastReceiver` 的频繁使用相比，`ContentProvider` 使用的频率并不高，因此有些插件化框架并不支持 `ContentProvider` 插件化。在讲到 `ContentProvider` 插件化之前仍旧要简单回顾一下 `ContentProvider` 的启动过程。

15.6.1 ContentProvider的启动过程回顾

`ContentProvider` 主要用于进程内和进程间的数据共享，它同样需要经过 `AMS` 来进行处理，`query` 方法到 `AMS` 的调用过程如图 15-12 所示。

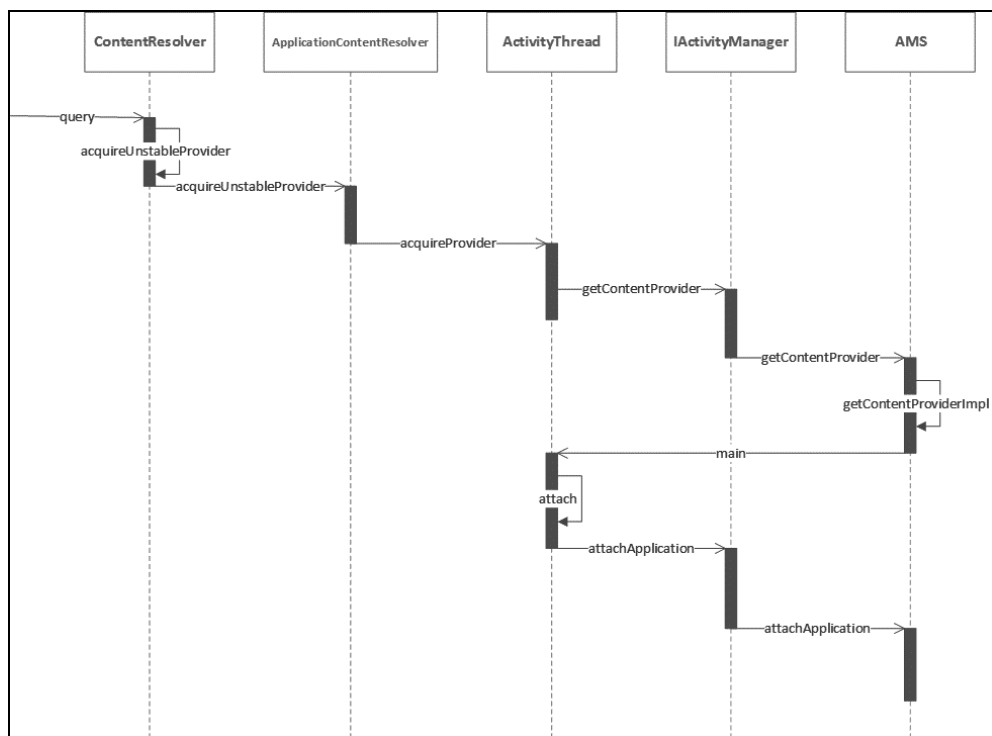


图 15-12 `query` 方法到 `AMS` 的调用过程

从图 15-12 可以看出 ContentProvider 的 query 方法会调用 ActivityThread 的 acquireProvider 方法，如下所示：

```
frameworks/base/core/java/android/app
```

```
public final IContentProvider acquireProvider(
    Context c, String auth, int userId, boolean stable) {
    final IContentProvider provider = acquireExistingProvider(c, auth, userId,
        stable);//1
    if (provider != null) {
        return provider;
    }
    ContentProviderHolder holder = null;
    try {
        holder = ActivityManager.getService().getContentProvider(
            getApplicationThread(), auth, userId, stable);//2
    } catch (RemoteException ex) {
        throw ex.rethrowFromSystemServer();
    }
    if (holder == null) {
        Slog.e(TAG, "Failed to find provider info for " + auth);
        return null;
    }
    holder = installProvider(c, holder, holder.info,
        true /*noisy*/, holder.noReleaseNeeded, stable);
    return holder.provider;
}
```

在注释 1 处的 acquireProvider 方法中会查询 mProviderMap 是否有目标 IContentProvider 存在，有则返回，没有就会调用注释 2 处的 IActivityManager 的 getContentProvider 方法，最终会调用 AMS 的 getContentProvider 方法获取 ContentProviderHolder（里面包含 IContentProvider 类型数据）。IContentProvider 是一个 Binder 对象，用于进程间通信，ContentProvider 的共享处理会委托给 IContentProvider 来处理。AMS 启动 Content Provider 的过程与 Activity 和 Service 类似，都是通过 ActivityThread 向 H 发送消息的，将代码逻辑运行在主线程中，最终调用 ContentProvider 的 onCreate 方法。具体的内容请查看 4.5 节，这里不再赘述。

15.6.2 VirtualApk的实现

ContentProvider 插件化的关键在于将 ContentProvider 插件共享给整个系统。和 Service 插件化类似，需要注册一个真正的 ContentProvider 作为代理 ContentProvider，并把这个代理 ContentProvider 共享给整个系统，对于插件 ContentProvider 的请求会全部交由代理

ContentProvider 处理并分发给对应的插件 ContentProvider。对于 ContentProvider 插件化的原理，这里不再像分析 Activity 和 Service 插件化那样去写一个小例子来实现，而是换一种形式，分析滴滴 VirtualApk 的 ContentProvider 插件化是如何实现的。相信有了前面的原理积累，你会发现再去分析第三方框架会轻松许多。

15.6.2.1 VirtualApk 初始化

要想更好地理解 VirtualApk 对于 ContentProvider 插件化的实现，需要先大概了解 VirtualApk 是如何初始化的。VirtualApk 给出了使用示例代码，在 MainActivity 的 onCreate 方法中调用了如下代码：

```
app/src/main/java/com/didi/virtualapk/MainActivity.java
```

```
this.loadPlugin(this);
```

MainActivity 的 loadPlugin 方法如下所示：

```
app/src/main/java/com/didi/virtualapk/MainActivity.java
```

```
private void loadPlugin(Context base) {
    PluginManager pluginManager = PluginManager.getInstance(base);//1
    File apk = new File(Environment.getExternalStorageDirectory(), "Test.apk");
    if (apk.exists()) {
        try {
            pluginManager.loadPlugin(apk);//2
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

在注释 1 处用于初始化插件化系统，在注释 2 处用于加载插件。PluginManager 的 getInstance 方法如下所示：

```
CoreLibrary/src/main/java/com/didi/virtualapk/PluginManager.java
```

```
public static PluginManager getInstance(Context base) {
    if (sInstance == null) {
        synchronized (PluginManager.class) {
            if (sInstance == null)
                sInstance = new PluginManager(base);
        }
    }
    return sInstance;
}
```



```

private PluginManager(Context context) {
    Context app = context.getApplicationContext();
    if (app == null) {
        this.mContext = context;//1
    } else {
        this.mContext = ((Application)app).getBaseContext();//2
    }
    prepare();
}
private void prepare() {
    Systems.sHostContext = getHostContext();//3
    this.hookInstrumentationAndHandler();//4
    if (Build.VERSION.SDK_INT >= 26) {
        this.hookAMSForO();
    } else {
        this.hookSystemService();
    }
}
}

```

PluginManager 的 getInstance 方法采用的是单例双重检查模式 (DCL)，在注释 3 处获取宿主的上下文 Context，值为 this.mContext，它在注释 1 和注释 3 处被赋值。注释 4 处用于 Hook Instrumentation 和 H，接着根据不同的系统版本 Hook IActivityManager，这些 Hook 原理在 15.4 节中都讲过。接着回头查看 MainActivity 的 loadPlugin 方法，在注释 2 处调用了 pluginManager 的 loadPlugin 方法：

CoreLibrary/src/main/java/com/didi/virtualapk/PluginManager.java

```

public void loadPlugin(File apk) throws Exception {
    ...
    LoadedPlugin plugin = LoadedPlugin.create(this, this.mContext, apk);//1
    if (null != plugin) {
        this.mPlugins.put(plugin.getPackageName(), plugin);
        plugin.invokeApplication();//2
    } else {
        throw new RuntimeException("Can't load plugin which is invalid: " +
            apk.getAbsolutePath());
    }
}
}

```

在注释 1 处调用了 LoadedPlugin 的 create 方法来创建 LoadedPlugin 对象，在注释 2 处创建插件的 Application。LoadedPlugin 的构造方法如下所示：

```

LoadedPlugin(PluginManager pluginManager, Context context, File apk) throws
PackageParser.PackageParserException {

```

```

        this.mPluginManager = pluginManager;
        this.mHostContext = context;
        ...
        this.mPluginContext = new PluginContext(this);//1
        this.mNativeLibDir = context.getDir(Constants.NATIVE_DIR,
            Context.MODE_PRIVATE);
        this.mResources = createResources(context, apk);
        this.mClassLoader = createClassLoader(context, apk, this.mNativeLibDir,
            context.getClassLoader());
        ...
        //ContentProvider 相关存储
        Map<String, ProviderInfo> providers = new HashMap<String, ProviderInfo>();
        Map<ComponentName, ProviderInfo> providerInfos = new HashMap<ComponentName,
            ProviderInfo>();
    }

```

LoadedPlugin 的构造方法代码比较多，主要用来创建一些类型的对象，比如 PackageInfo、Resources、ClassLoader 等，还有创建存储四大组件相关的数据结构，比如 Map 等。需要主意的是，注释 1 处会创建插件的上下文 PluginContext。关于 VirtualApk 初始化就讲到这里，我们接着回到 ContentProvider 插件化这一话题。

15.6.2.2 启动代理 ContentProvider

在 MainActivity 的 onClick 方法中会测试 ContentProvider，如下所示：

app/src/main/java/com/didi/virtualapk/MainActivity.java

```

public void onClick(View v) {
    if (v.getId() == R.id.button) {
        final String pkg = "com.didi.virtualapk.demo";
        ...
        // test ContentProvider
        Uri bookUri = Uri.parse("content://com.didi.virtualapk.demo.book.
            provider/book");
        LoadedPlugin plugin = PluginManager.getInstance(this).getLoadedPlugin
            (pkg);
        bookUri = PluginContentResolver.wrapperUri(plugin, bookUri);
        Cursor bookCursor = getContentResolver().query(bookUri, new String[]
            {"_id", "name"}, null, null, null);//1
        ...
    } else if (v.getId() == R.id.about) {
        showAbout();
    }
}

```

在 Activity 启动前 VirtualApk 就通过 Hook Instrumentation 的形式,在 VAINstrumentation 的 callActivityOnCreate 方法中用 PluginContext 替换了 ContextWrapper 的成员变量 mBase, mBase 是 Context 类型的,不了解 mBase 的请查看第 5 章。注释 1 处的 getContentResolver 实际调用的是 PluginContext 的 getContentResolver 方法:

CoreLibrary/src/main/java/com/didi/virtualapk/internal/PluginContext.java

```
@Override
public ContentResolver getContentResolver() {
    return new PluginContentResolver(getHostContext());
}
```

在 getContentResolver 方法中只创建了 PluginContentResolver 并传入了宿主的 Context, 因此当我们调用 getContentResolver 的 query 方法时,实际上是调用了 PluginContentResolver 的 query 方法。根据图 15-12 所示可知, query 方法会调用 ContentResolver 的 acquireUnstableProvider 方法, PluginContentResolver 复写了 acquireUnstableProvider 方法:

CoreLibrary/src/main/java/com/didi/virtualapk/internal/PluginContentResolver.java

```
protected IContentProvider acquireUnstableProvider(Context context, String auth) {
    try {
        if (mPluginManager.resolveContentProvider(auth, 0) != null) { //1
            return mPluginManager.getIContentProvider();
        }
        return (IContentProvider) sAcquireUnstableProvider.invoke(mBase, context,
            auth);
    } catch (Exception e) {
        e.printStackTrace();
    }
    return null;
}
```

在注释 1 处会查找插件中是否有匹配的 ContentProvider, 如果没有就调用系统 ContentResolver 的 acquireUnstableProvider 方法, 有的话会调用 PluginManager 的 getIContentProvider 方法, 如下所示:

```
public synchronized IContentProvider getIContentProvider() {
    if (mIContentProvider == null) {
        hookIContentProviderAsNeeded();
    }
    return mIContentProvider;
}
```

hookIContentProviderAsNeeded 方法如下所示:

CoreLibrary/src/main/java/com/didi/virtualapk/PluginManager.java

```

private void hookIContentProviderAsNeeded() {
    //获取插件 ContentResolver 的 Uri
    Uri uri = Uri.parse(PluginContentResolver.getUri(mContext)); //1
    //得到 IContentProvider
    mContext.getContentResolver().call(uri, "wakeup", null, null); //2
    try {
        Field authority = null;
        Field mProvider = null;
        //反射得到 ActivityThread 实例
        ActivityThread activityThread = (ActivityThread) ReflectUtil.
            getActivityThread(mContext);
        Map mProviderMap = (Map) ReflectUtil.getField(activityThread.getClass(),
            "mProviderMap"); //3
        Iterator iter = mProviderMap.entrySet().iterator();
        while (iter.hasNext()) {
            Map.Entry entry = (Map.Entry) iter.next();
            Object key = entry.getKey();
            Object val = entry.getValue();
            String auth;
            if (key instanceof String) {
                auth = (String) key;
            } else {
                if (authority == null) {
                    authority = key.getClass().getDeclaredField("authority");
                    authority.setAccessible(true);
                }
                auth = (String) authority.get(key);
            }
            if (auth.equals(PluginContentResolver.getAuthority(mContext))) {
                if (mProvider == null) {
                    mProvider = val.getClass().getDeclaredField("mProvider");
                    mProvider.setAccessible(true);
                }
                IContentProvider rawProvider = (IContentProvider) mProvider.
                    get(val);
                IContentProvider proxy = IContentProviderProxy.newInstance
                    (mContext, rawProvider); //4
                mIContentProvider = proxy; //5
                Log.d(TAG, "hookIContentProvider succeed : " + mIContentProvider);
                break;
            }
        }
    } catch (Exception e) {

```

```

        e.printStackTrace();
    }
}

```

从方法名就可以看出，这个方法是用来 Hook `IContentProvider` 的。在注释 1 处获得代理 `ContentResolver` 的 `Uri`。在注释 2 处调用 `ContentResolver` 的 `call` 方法，`mContext` 是宿主的 `Context`，因此这里调用的是宿主 `ContentResolver` 的 `call` 方法，用于得到 `IContentProvider`。在注释 3 处获取 `ActivityThread` 的 `mProviderMap`，接下来遍历 `mProviderMap`，找到匹配的 `IContentProvider`，在注释 5 处用代理 `IContentProviderProxy` 替换 `IContentProvider`，完成 Hook `IContentProvider`。`IContentProviderProxy` 的 `wrapperUri` 方法实现了替换 `Uri` 的操作，如下所示：

CoreLibrary/src/main/java/com/didi/virtualapk/delegate/IContentProviderProxy.java

```

private void wrapperUri(Method method, Object[] args) {
    ...
    PluginManager pluginManager = PluginManager.getInstance(mContext);
    ProviderInfo info = pluginManager.resolveContentProvider(uri.getAuthority(), 0);
    if (info != null) {
        String pkg = info.packageName;
        LoadedPlugin plugin = pluginManager.getLoadedPlugin(pkg);
        String pluginUri = Uri.encode(uri.toString());
        StringBuilder builder = new StringBuilder(PluginContentResolver.getUri(mContext)); //1
        builder.append("/?plugin=" + plugin.getLocation());
        builder.append("&pkg=" + pkg);
        builder.append("&uri=" + pluginUri);
        Uri wrapperUri = Uri.parse(builder.toString()); //2
        if (method.getName().equals("call")) {
            bundleInCallMethod.putString(KEY_WRAPPER_URI, wrapperUri.toString());
        } else {
            args[index] = wrapperUri; //3
        }
    }
}
}

```

在注释 1 处获取插件的 `Uri` 并封装成 `StringBuilder`，接下来对 `StringBuilder` 进行拼接，在注释 2 处得到一个新的 `Uri`，在注释 3 处替换 `Uri`，这样我们启动一个插件 `ContentProvider` 时会先启动代理 `ContentProvider`。

15.6.2.3 代理分发

代理 `ContentProvider` 已经在 `AndroidManifest.xml` 中注册了，如下所示：

CoreLibrary/src/main/AndroidManifest.xml

```
<provider
    android:name="com.didi.virtualapk.delegate.RemoteContentProvider"
    android:authorities="${applicationId}.VirtualAPK.Provider"
    android:process=":daemon" />
```

当我们调用 ContentProvider 的 query 方法时，实际会调用 RemoteContentProvider 的 query 方法：

CoreLibrary/src/main/java/com/didi/virtualapk/delegate/RemoteContentProvider.java

```
@Override
public Cursor query(Uri uri, String[] projection, String selection, String[]
selectionArgs, String sortOrder) {
    ContentProvider provider = getContentProvider(uri); //1
    Uri pluginUri = Uri.parse(uri.getQueryParameter(KEY_URI));
    if (provider != null) {
        return provider.query(pluginUri, projection, selection, selectionArgs,
            sortOrder);
    }
    return null;
}
```

在注释 1 处会通过 RemoteContentProvider 的 getContentProvider 方法得到一个 ContentProvider，并调用它的 query 方法，RemoteContentProvider 的 getContentProvider 方法如下所示：

CoreLibrary/src/main/java/com/didi/virtualapk/delegate/RemoteContentProvider.java

```
private ContentProvider getContentProvider(final Uri uri) {
    final PluginManager pluginManager = PluginManager.getInstance(getContext());
    Uri pluginUri = Uri.parse(uri.getQueryParameter(KEY_URI));
    final String auth = pluginUri.getAuthority();
    ContentProvider cachedProvider = sCachedProviders.get(auth); //1
    if (cachedProvider != null) {
        return cachedProvider;
    }
    synchronized (sCachedProviders) {
        LoadedPlugin plugin = pluginManager.getLoadedPlugin(uri.getQueryParameter
            (KEY_PKG));
        if (plugin == null) {
            try {
                pluginManager.loadPlugin(new File(uri.getQueryParameter(KEY_
PLUGIN))); //2
            } catch (Exception e) {
```

```

        e.printStackTrace();
    }
}
final ProviderInfo providerInfo = pluginManager.resolveContentProvider(auth,
0);//3
if (providerInfo != null) {
    RunUtil.runOnUiThread(new Runnable() {
        @Override
        public void run() {
            try {
                LoadedPlugin loadedPlugin = pluginManager.getLoadedPlugin(uri.
                    getQueryParameter(KEY_PKG));
                ContentProvider contentProvider = (ContentProvider) Class.
forName(providerInfo.name).newInstance();//4
                contentProvider.attachInfo(loadedPlugin.getPluginContext(),
                    providerInfo);//5
                sCachedProviders.put(auth, contentProvider);
            } catch (Exception e) {
                e.printStackTrace();
            }
        }
    }, true);
    return sCachedProviders.get(auth);
}
}
return null;
}

```

传入的参数是插件 uri 并解析为 auth，在注释 1 处会从缓存中读取是否有匹配 auth 的插件 ContentProvider，没有的话就会在注释 2 处加载插件 APK，并从已加载的 APK 中得到匹配 auth 的 ProviderInfo，根据这个 ProviderInfo 就可以在注释 4 处创建插件 ContentProvider。在注释 5 处调用插件 ContentProvider 的 attachInfo 方法，其内部会为插件 ContentProvider 配置参数并调用它的 onCreate 方法，这样插件 ContentProvider 就启动了。最后将这个新创建的插件 ContentProvider 加入缓存中，避免重复创建插件 ContentProvider。

15.7 BroadcastReceiver的插件化

BroadcastReceiver 的注册分为两种，分别是静态注册和动态注册，在 4.4 节介绍了 BroadcastReceiver 的动态注册，动态注册通过 AMS 来完成，动态注册的信息会存在 AMS

中。静态注册需要在 AndroidManifest.xml 中注册，应用在安装时，PackageManagerService (PMS) 会调用 PackageParser 的 parsePackage 方法来解析 APK，通过解析 APK 中的 AndroidManifest.xml 文件的标签得到 APK 中的各种信息并封装成相应的信息类，比如 ApplicationInfo、ProviderInfo 和 ActivityInfo 等，因此动态注册的信息会存在 PMS 中。

15.7.1 广播插件化思路

静态注册的 BroadcastReceiver 会在 AndroidManifest.xml 中设置<intent-filter>标签，BroadcastReceiver 根据这个标签中的值来接收“感兴趣”的广播。如果采用类似 Activity 插件化的 Hook IActivityManager 方案，用一个占坑 BroadcastReceiver 来接收广播是不可行的，因为我们无法预料插件中静态注册的 BroadcastReceiver 的<intent-filter>标签，这样占坑 BroadcastReceiver 无法接收到“感兴趣”的广播。静态注册的 BroadcastReceiver 的<intent-filter>标签无法动态设置，但是动态注册的 BroadcastReceiver 是可以动态设置 IntentFilter 的，讲到这里我们得到了一个新思路，那就是将静态注册的 BroadcastReceiver 全部转换为动态注册来处理，虽然静态和动态的 BroadcastReceiver 的生命周期不同，但是为了实现插件化，这个缺点显然不是关键问题。

15.7.2 VirtualApk的实现

在 LoadedPlugin 的构造方法中实现静态注册的 BroadcastReceiver 转换为动态注册，如下所示：

CoreLibrary/src/main/java/com/didi/virtualapk/internal/LoadedPlugin.java

```
LoadedPlugin(PluginManager pluginManager, Context context, File apk) throws
PackageParser.PackageParserException {
    ...
    Map<ComponentName, ActivityInfo> receivers = new HashMap<ComponentName,
    ActivityInfo>();//1
    for (PackageParser.Activity receiver : this.mPackage.receivers) {
        receivers.put(receiver.getComponentName(), receiver.info);//2
        try {
            BroadcastReceiver br = BroadcastReceiver.class.
            cast(getClassLoader().loadClass(receiver.getComponentName().
            getClassName()).newInstance());//3
            for (PackageParser.ActivityIntentInfo aii : receiver.intents) {
                this.mHostContext.registerReceiver(br, aii);//4
            }
        } catch (Exception e) {
```



```

        e.printStackTrace();
    }
}
this.mReceiverInfos = Collections.unmodifiableMap(receivers);
this.mPackageInfo.receivers = receivers.values().toArray(new ActivityInfo
[receivers.size()]);
}

```

注释 1 处的 receivers 用于存储插件中静态注册的 BroadcastReceiver 信息，有的读者可能会有疑问，为什么 receivers 的泛型类型会是 ActivityInfo，这是因为 PackageParser 在解析 AndroidManifest.xml 时把 <receiver> 标签当作 <activity> 标签处理了，因此解析得到的 BroadcastReceiver 信息会存储在 ActivityInfo 中。在注释 2 处将插件 BroadcastReceiver 信息存储在 receivers 中。注释 3 处的代码有点长，它分为两个步骤，步骤 1 根据插件 BroadcastReceiver 的类名，用 ClassLoader 加载并创建对象（类型为 Object）；步骤 2 是将 Object 转换为 BroadcastReceiver 类型。在注释 4 处调用宿主 Context 的 registerReceiver 方法来完成插件 BroadcastReceiver 的注册。

15.8 资源的插件化

在上一章讲解热修复原理时讲解了资源修复，资源修复与 AssetManager 是有关的，同样地，资源的插件化也是如此。我们先来看系统的资源是如何加载的。

15.8.1 系统资源加载

启动 Activity 时会调用 performLaunchActivity 方法，其内部会调用 LoadedApk 的 makeApplication 方法：

```

frameworks/base/core/java/android/app/LoadedApk.java

public Application makeApplication(boolean forceDefaultAppClass,
    Instrumentation instrumentation) {
    ...
    ContextImpl appContext = ContextImpl.createAppContext(mActivityThread,
        this);
    app = mActivityThread.mInstrumentation.newApplication(
        cl, appClass, appContext); //1
    appContext.setOuterContext(app);
    ...
}

```

在注释 1 处创建 Application，ContextImpl 的 createAppContext 方法用于创建应用的 Context：

frameworks/base/core/java/android/app/ContextImpl.java

```
static ContextImpl createAppContext(ActivityThread mainThread, LoadedApk packageInfo) {
    if (packageInfo == null) throw new IllegalArgumentException("packageInfo");
    ContextImpl context = new ContextImpl(null, mainThread, packageInfo, null, null,
        null, 0, null);
    context.setResources(packageInfo.getResources()); //1
    return context;
}
```

在注释 1 处调用 LoadedApk 的 getResources 方法得到 Resources，并将 Resources 赋值给 ContextImpl。LoadedApk 的 getResources 方法如下所示：

frameworks/base/core/java/android/app/LoadedApk.java

```
public Resources getResources() {
    ...
    mResources = ResourceManager.getInstance().getResources(null, mResDir,
        splitPaths, mOverlayDirs, mApplicationInfo.sharedLibraryFiles,
        Display.DEFAULT_DISPLAY, null, getCompatibilityInfo(),
        getClassLoader());
}
return mResources;
}
```

getResources 方法会调用 ResourceManager 的 getResources 方法，其内部会返回 getOrCreateResources 方法：

```
private @Nullable Resources getOrCreateResources(@Nullable IBinder activityToken,
    @NonNull ResourcesKey key, @NonNull ClassLoader classLoader) {
    ...
    ResourcesImpl resourcesImpl = createResourcesImpl(key); //1
    if (resourcesImpl == null) {
        return null;
    }
    synchronized (this) {
        ...
        final Resources resources;
        if (activityToken != null) {
            resources = getOrCreateResourcesForActivityLocked(activityToken,
                classLoader, resourcesImpl, key.mCompatInfo); //2
        } else {
            resources = getOrCreateResourcesLocked(classLoader, resourcesImpl,
                key.mCompatInfo);
        }
    }
}
```

```

    }
    return resources;
}
}

```

在注释 1 处创建 `ResourcesImpl`，它用于具体实现 `Resources`，在 `Resources` 创建后，会调用 `Resources` 的 `setImpl` 方法将 `ResourcesImpl` 设置进去。在注释 2 处创建 `Resources`。`createResourcesImpl` 方法如下所示：

frameworks/base/core/java/android/app/ResourcesManager.java

```

private @Nullable ResourcesImpl createResourcesImpl(@NonNull ResourcesKey key) {
    final DisplayAdjustments daj = new DisplayAdjustments(key.mOverride
        Configuration);
    daj.setCompatibilityInfo(key.mCompatInfo);
    final AssetManager assets = createAssetManager(key);//1
    if (assets == null) {
        return null;
    }
    final DisplayMetrics dm = getDisplayMetrics(key.mDisplayId, daj);
    final Configuration config = generateConfig(key, dm);
    final ResourcesImpl impl = new ResourcesImpl(assets, dm, config, daj);//2
    if (DEBUG) {
        Slog.d(TAG, "- creating impl=" + impl + " with key: " + key);
    }
    return impl;
}

```

在注释 1 处创建 `AssetManager`，在注释 2 处新建 `ResourcesImpl` 对象，并将 `AssetManager` 作为参数传进去，这是因为 `Resources` 会依赖 `AssetManager` 来加载资源。

15.8.2 VirtualApk实现

资源的插件化方案主要有两种：一种是合并资源方案，将插件的资源全部添加到宿主的 `Resources` 中，这种方案插件可以访问宿主的资源。另一种是构建插件资源方案，每个插件都构造出独立的 `Resources`，这种方案插件不可以访问宿主资源。`VirtualApk` 采用了以上两种方案，具体的代码逻辑在 `LoadedPlugin` 中，如下所示：

CoreLibrary/src/main/java/com/didi/virtualapk/internal/LoadedPlugin.java

```

@WorkerThread
private static Resources createResources(Context context, File apk) {
    if (Constants.COMBINE_RESOURCES) {
        Resources resources = ResourcesManager.createResources(context, apk.
            getAbsolutePath());
    }
}

```

```

        ResourceManager.hookResources(context, resources); //1
        return resources;
    } else {
        Resources hostResources = context.getResources();
        AssetManager assetManager = createAssetManager(context, apk); //2
        return new Resources(assetManager, hostResources.getDisplayMetrics(),
            hostResources.getConfiguration()); //3
    }
}

```

createResources 方法用于创建 Resources, 如果是合并资源方案, 会调用 ResourceManager 的 createResources 方法, 其内部会先得到包含宿主资源的 AssetManager, 再通过反射调用 AssetManager 的 addAssetPath 来添加插件资源, 返回新的 Resources, 在注释 1 处通过 Hook 的方式用新的 Resources 替换此前的 Resources。如果是构建插件资源方案, 会在注释 2 处先创建 AssetManager, 再创建 Resources 并将 AssetManager 作为参数传进去。createAssetManager 方法如下所示:

CoreLibrary/src/main/java/com/didi/virtualapk/internal/LoadedPlugin.java

```

private static AssetManager createAssetManager(Context context, File apk) {
    try {
        AssetManager am = AssetManager.class.newInstance();
        ReflectUtil.invoke(AssetManager.class, am, "addAssetPath", apk.
            getAbsolutePath());
        return am;
    } catch (Exception e) {
        e.printStackTrace();
        return null;
    }
}

```

首先动态创建 AssetManager, 再反射调用 AssetManager 的 addAssetPath 方法来加载插件, 这个 AssetManager 只包含了插件的资源, 因此 createResources 方法的注释 3 处新创建的 Resources 是插件的资源。VirtualApk 实现就讲到这里, 关于系统版本和手机型号适配的问题这里没有提到, 想了解的读者请自行阅读 VirtualApk 的源码。

15.9 so的插件化

so 的插件化可以结合第 13 章讲到的 so 热修复来学习, so 热修复主要有两种方案:

- 将 so 补丁插入到 NativeLibraryElement 数组的前部，让 so 补丁的路径先被返回和加载。
- 调用 System 的 load 方法来接管 so 的加载入口。

so 的插件化的方案和 so 热修复第一种方案类似，简单来说就是将 so 插件插入到 NativeLibraryElement 数组中，并且将存储 so 插件的文件添加到 nativeLibraryDirectories 集合中就可以了。我们来查看 VirtualApk 实现，在 LoadedPlugin 的构造方法中会调用 createClassLoader 方法：

CoreLibrary/src/main/java/com/didi/virtualapk/internal/LoadedPlugin.java

```
private static ClassLoader createClassLoader(Context context, File apk, File
libsDir, ClassLoader parent) {
    File dexOutputDir = context.getDir(Constants.OPTIMIZE_DIR, Context.MODE_
PRIVATE);
    String dexOutputPath = dexOutputDir.getAbsolutePath();
    DexClassLoader loader = new DexClassLoader(apk.getAbsolutePath(),
dexOutputPath, libsDir.getAbsolutePath(), parent);//1
    if (Constants.COMBINE_CLASSLOADER) {
        try {
            DexUtil.insertDex(loader);
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
    return loader;
}
```

在注释 1 处创建用于加载插件的 DexClassLoader，并作为参数传入 DexUtil 的 insertDex 方法中：

CoreLibrary/src/main/java/com/didi/virtualapk/utlis/DexUtil.java

```
public static void insertDex(DexClassLoader dexClassLoader) throws Exception {
    Object baseDexElements = getDexElements(getPathList(getPathClassLoader()));
    Object newDexElements = getDexElements(getPathList(dexClassLoader));
    Object allDexElements = combineArray(baseDexElements, newDexElements);
    Object pathList = getPathList(getPathClassLoader());
    ReflectUtil.setField(pathList.getClass(), pathList, "dexElements", allDex
Elements);//1
    insertNativeLibrary(dexClassLoader);
}
```

insertDex 方法会将宿主和插件的 DexElements 合并得到 allDexElements，并通过反射用

allDexElements 替换 dexElements。关于注释 1 处的用法和 14.5.2.3 节讲到的 FieldUtil.setField 方法类似。so 的插件化的核心代码在 insertNativeLibrary 方法中，如下所示：

CoreLibrary/src/main/java/com/didi/virtualapk/utlis/DexUtil.java

```
private static synchronized void insertNativeLibrary(DexClassLoader dexClassLoader)
throws Exception {
    if (sHasInsertedNativeLibrary) { //1
        return;
    }
    sHasInsertedNativeLibrary = true;
    Object basePathList = getPathList(getPathClassLoader()); //2
    //Android 版本大于 5.1
    if (Build.VERSION.SDK_INT > Build.VERSION_CODES.LOLLIPOP_MR1) {
        List<File> nativeLibraryDirectories = (List<File>) ReflectUtil.getField(
            (basePathList.getClass(), basePathList, "nativeLibraryDirectories")); //3
        nativeLibraryDirectories.add(Systems.getContext().getDir(Constants.
            NATIVE_DIR, Context.MODE_PRIVATE)); //4
        Object baseNativeLibraryPathElements = ReflectUtil.getField(basePathList.
            getClass(), basePathList, "nativeLibraryPathElements"); //5
        final int baseArrayLength = Array.getLength(baseNativeLibraryPathElements);
        Object newPathList = getPathList(dexClassLoader);
        Object newNativeLibraryPathElements = ReflectUtil.getField(newPathList.
            getClass(), newPathList, "nativeLibraryPathElements"); //6
        Class<?> elementClass = newNativeLibraryPathElements.getClass().
            getComponentType(); //7
        Object allNativeLibraryPathElements = Array.newInstance(elementClass,
            baseArrayLength + 1);
        System.arraycopy(baseNativeLibraryPathElements, 0,
            allNativeLibraryPathElements, 0, baseArrayLength); //8
        Field soPathField;
        if (Build.VERSION.SDK_INT >= 26) {
            soPathField = elementClass.getDeclaredField("path");
        } else {
            soPathField = elementClass.getDeclaredField("dir");
        }
        soPathField.setAccessible(true);
        final int newArrayLength = Array.getLength(newNativeLibraryPathElements);
        for (int i = 0; i < newArrayLength; i++) {
            Object element = Array.get(newNativeLibraryPathElements, i);
            String dir = ((File)soPathField.get(element)).getAbsolutePath();
            if (dir.contains(Constants.NATIVE_DIR)) {
                Array.set(allNativeLibraryPathElements, baseArrayLength, element);
                break;
            }
        }
    }
}
```

```

        }
    }
    ReflectUtil.setField(basePathList.getClass(), basePathList,
        "nativeLibraryPathElements", allNativeLibraryPathElements);
} else {
    ...
}
}

```

注释 1 处用于避免重复插入 so。在注释 2 处获取的 `basePathList` 指的是宿主的 `PathList`。接下来对不同的 Android 版本会有不同的处理，这里只讨论 Android 版本大于 5.1 的情况。在注释 3 处得到宿主存储 so 文件的 List 集合 `nativeLibraryDirectories`，紧接着在注释 4 处将插件存储 so 文件添加到 `nativeLibraryDirectories` 中。在注释 5 和注释 6 处分别获取宿主的 `NativeLibraryElement`（`baseNativeLibraryPathElements`）和插件的 `NativeLibraryElement`（`newNativeLibraryPathElements`）。在注释 7 处得到插件的 `newNativeLibraryPathElements` 的类型，并创建这个类型的数组 `allNativeLibraryPathElements`，在注释 8 处将 `baseNativeLibraryPathElements` 复制到 `allNativeLibraryPathElements` 中。遍历 `newNativeLibraryPathElements`，将 so 添加到 `allNativeLibraryPathElements` 中。最后通过反射用 `allNativeLibraryPathElements` 替换 `nativeLibraryPathElements`，这样就完成了 so 的插件化。

15.10 本章小结

本章介绍了插件化的产生、插件化框架对比、四大组件的插件化、资源和 so 的插件化等。插件化是一个很庞大的知识体系，用一章的内容只能介绍了部分的插件化原理，比如插件的加载机制就没有讲到，插件的加载机制方案主要有两种，一种是 Hook ClassLoader，另一种是委托给系统的 ClassLoader 帮忙加载，无论是哪一种方案，原理都离不开第 12 章讲解的内容。另外本章介绍的插件化原理是建立在主流框架之上的，如果想要掌握全部的插件化原理，请阅读表 15-1 中列出的全部插件化框架源码。

第 16 章

绘制优化

关联章节：第 10 章 Java 虚拟机；第 11 章 Dalvik 和 ART；

随着 Android 应用开发日趋成熟，中大型应用越来越多，应用市场竞争也越来越激烈。应用除了自身的业务和功能吸引用户外，还需要提供用户更好的体验，这个更好的体验就对应用性能上有更高的要求。运行 Android 系统的手机，虽然配置在不断提升，但仍旧无法和 PC 相比，无法做到 PC 那样拥有超大的内存以及高性能的 CPU，因此在开发 Android 应用程序时也不可能无限制地使用 CPU 和内存，如果对 CPU 和内存使用不当也会造成应用的卡顿和内存溢出等问题。体验以及设备的限制导致性能优化越来越受到企业的重视，因此性能优化也成为了 Android 应用开发者必备的技能之一。

性能优化是一个很庞大的知识体系，它包括绘制优化、内存优化、电量优化、启动优化、存储优化、流量优化、图片优化和 APK 优化等，这些内容完全可以写一本书。结合本书的篇幅和本书前面的章节内容（第 10 章、第 11 章），本书将介绍其中的绘制优化和内存优化。

16.1 绘制性能分析

Android 应用需要将自己的界面展示给用户，用户会和界面进行交互，界面的流畅度至关重要，这一节我们就来学习绘制性能分析，首先讲解绘制原理，接着介绍绘制性能分析的工具：Profile GPU Rendering、Systrace 和 Traceview。

16.1.1 绘制原理

View 的绘制流程有 3 个步骤，分别是 measure、layout 和 draw，它们主要运行在系统的应用框架层，而真正将数据渲染到屏幕上的则是系统 Native 层的 SurfaceFlinger 服务来完成的。

绘制过程主要由 CPU 来进行 Measure、Layout、Record、Execute 的数据计算工作，GPU 负责栅格化、渲染。CPU 和 GPU 是通过图形驱动层来进行连接的，图形驱动层维护了一个队列，CPU 将 display list 添加到该队列中，这样 GPU 就可以从这个队列中取出数据进行绘制。

说到绘制性能就需要提到帧数这个概念。帧数就是在 1 秒时间里传输的图片的量，也可以理解为图形处理器每秒钟能够刷新几次，通常用 FPS (Frames Per Second) 表示。每一帧其实就是静止的图像，通过快速连续地显示帧便形成了运动的假象。最简单的举例就是我们玩游戏时，如果画面在 60fps 则不会感觉到卡顿，如果低于 60fps，比如 50fps 则会感觉到卡顿。这是因为人类的大脑会不断接收并处理眼球看到的信息，单位时间内越多的帧被处理，越能有效地被大脑识别，大脑能感知的最小帧数在 10fps~12fps，这个时候大脑就分不清这个图像是静止的还是变化的。

要想画面保持在 60fps，需要屏幕在 1 秒内刷新 60 次，也就是每 16.6667ms 刷新一次（绘制时长在 16ms 以内），如图 16-1 所示。

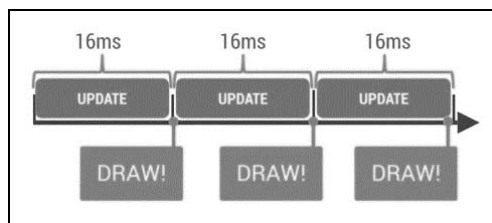


图 16-1 画面保持在 60fps

Android 系统每隔 16ms 发出 VSYNC 信号，触发对 UI 进行渲染，如果每次渲染都成功，这样就能够达到流畅的画面所需要的 60fps，那什么是 VSYNC 呢？VSYNC 是 Vertical Synchronization（垂直同步）的缩写，是一种定时中断，一旦收到 VSYNC 信号，CPU 就开始处理各帧数据。如果某个操作要花费 24ms，这样系统在得到 VSYNC 信号时无法进行正常的渲染，会发生丢帧。用户会在 32ms 中看到同一帧的画面，如图 16-2 所示。

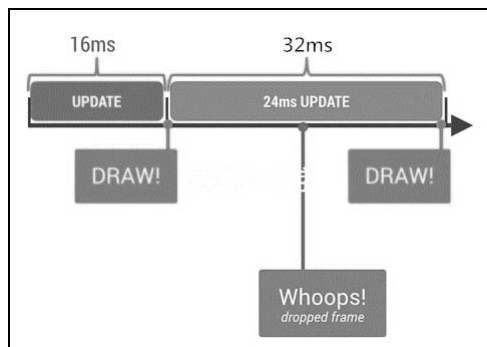


图 16-2 画面无法保持在 60fps

产生卡顿原因有很多，主要有以下几点：

- 布局 Layout 过于复杂，无法在 16ms 内完成渲染。
- 同一时间动画执行的次数过多，导致 CPU 或 GPU 负载过重。
- View 过度绘制，导致某些像素在同一帧时间内被绘制多次。
- 在 UI 线程中做了稍微耗时的操作。
- GC 回收时暂停时间过长或者频繁的 GC 产生大量的暂停时间。

为了解决上述的问题，除了我们要在写代码时要注意外，也可以借助一些工具来分析和解决卡顿问题。

16.1.2 Profile GPU Rendering

Profile GPU Rendering 是 Android 4.1 系统提供的开发辅助功能，我们可以在开发者选项中打开这一功能，如图 16-3 所示。

我们单击 Profile GPU Rendering 选项并选择 On screen as bars 即开启 Profile GPU Rendering 功能。接着屏幕会显示出彩色的柱状图，如图 16-4 所示。

图中横轴代表时间，纵轴表示某一帧的耗时。绿色的横线为警戒线，超过这条线则意味着时长超过了 16ms，尽量要保证垂直的彩色柱状图保持在绿线下面。这些垂直的彩色柱状图代表着一帧，不同颜色的彩色柱状图代表不同的含义。

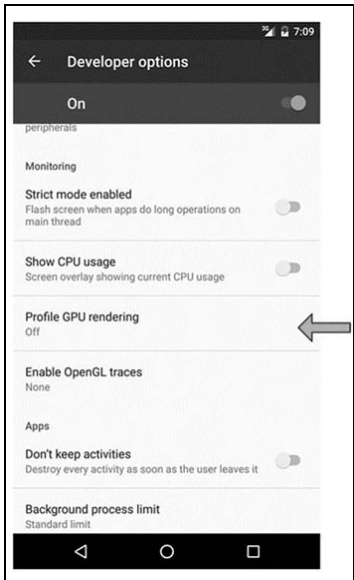


图 16-3 开发者选项

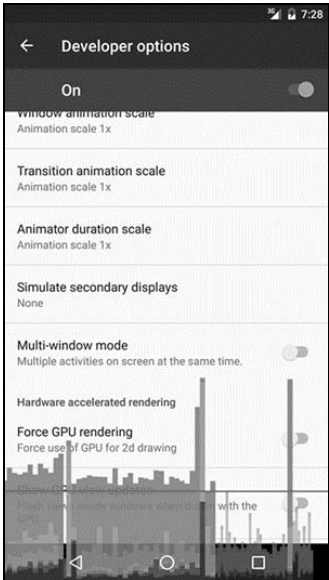


图 16-4 彩色的柱状图

- 橙色代表处理的时间，是 CPU 告诉 GPU 渲染一帧的地方，这是一个阻塞调用，因为 CPU 会一直等待 GPU 发出接到命令的回复，如果橙色柱状图很高，则表明 GPU 很繁忙。
- 红色代表执行的时间，这部分是 Android 进行 2D 渲染 Display List 的时间。如果红色柱状图很高，可能由于重新提交了视图而导致的。还有复杂的自定义 View 也会导致红的柱状图变高。
- 蓝色代表测量绘制的时间，也就是需要多长时间去创建和更新 DisplayList。如果蓝色柱状图很高，可能需要重新绘制，或者 View 的 onDraw 方法处理事情太多。

在 Android 6.0 中，有更多的颜色被加了进来，Android 6.0 颜色含义如图 16-5 所示。

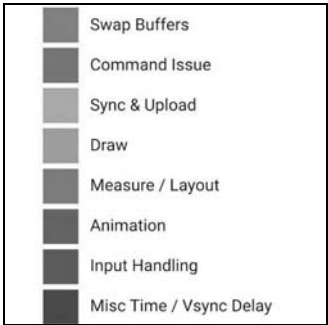


图 16-5 Android 6.0 颜色含义

由于书的印刷不是彩印，因此读者无法区分图 16-4 和 16-5 中的颜色，这一点可以参考我的博客文章：<http://liuwangshu.cn/application/performance/draw-1-performance.html>。

下面分别介绍它们的含义。

- Swap Buffers：表示处理的时间，和上面讲到的橙色一样。
- Command Issue：表示执行的时间，和上面讲到的红色一样。
- Sync & Upload：表示的是准备当前界面上有待绘制的图片所耗费的时间，为了减少该段区域的执行时间，我们可以减少屏幕上的图片数量或者缩小图片的大小。
- Draw：表示测量和绘制视图列表所需要的时间，和上面讲到的蓝色一样。
- Measure/Layout：表示布局的 onMeasure 与 onLayout 所花费的时间，一旦时间过长，就需要仔细检查自己的布局是不是存在严重的性能问题。
- Animation：表示计算执行动画所需要花费的时间，包含的动画有 ObjectAnimator、ViewPropertyAnimator、Transition 等。一旦这里的执行时间过长，就需要检查是不是使用了非官方的动画工具或者检查动画执行的过程中是不是触发了读/写操作等。
- Input Handling：表示系统处理输入事件所耗费的时间，粗略等于对事件处理方法所执行的时间。一旦执行时间过长，意味着在处理用户的输入事件的地方执行了复杂的操作。
- Misc Time/Vsync Delay：表示在主线程中执行了太多的任务，导致 UI 渲染跟不上 VSYNC 的信号而出现掉帧的情况。

Profile GPU Rendering 可以找到渲染有问题的界面，但是想要修复的话，只依赖 Profile GPU Rendering 是不够的，可以用另一个工具 Hierarchy Viewer 来查看布局层次和每个 View 所花的时间，这个工具会在后面进行介绍。

16.1.3 Systrace

Systrace 是 Android 4.1 中新增的性能数据采样和分析工具，它可以帮助开发者收集 Android 关键子系统（SurfaceFlinger、WMS 等 Framework 部分关键模块、服务，View 体系系统等）的运行信息。Systrace 的功能包括跟踪系统的 I/O 操作、内核工作队列、CPU 负载以及 Android 各个子系统的运行状况等。对于 UI 显示性能，比如动画播放不流畅、渲染卡顿等问题提供了分析数据。

16.1.3.1 使用 Systrace

Systrace 跟踪的设备要在 Android 4.1 以上版本中使用，对于 Android 4.3 版本之前和 4.3

版本之后使用上有点区别，现在也很少有人用 Android 4.3 之前的版本了，因此这里只讲 Android 4.3 以后版本的使用方法。Systrace 可以在 DDMS 上使用，可以使用命令行来使用，也可以在代码中进行跟踪。接下来分别来介绍这三种方式。

1. 在 DDMS 中使用 Systrace

- (1) 首先我们要打开 Android Studio 的 Tool 中的 Android Device Monitor，并连接手机。
- (2) 单击 Systrace 按钮进入抓取设置界面，如图 16-6 所示。

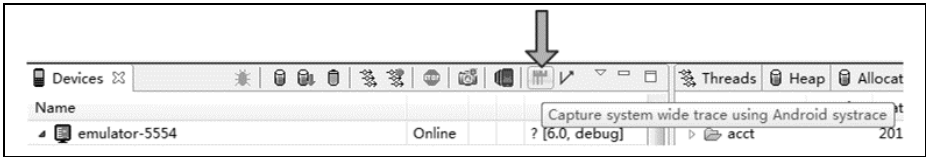


图 16-6 单击 Systrace 按钮

抓取设置界面可以设置跟踪的时间，以及 trace 文件输出的地址等内容，如图 16-7 所示。

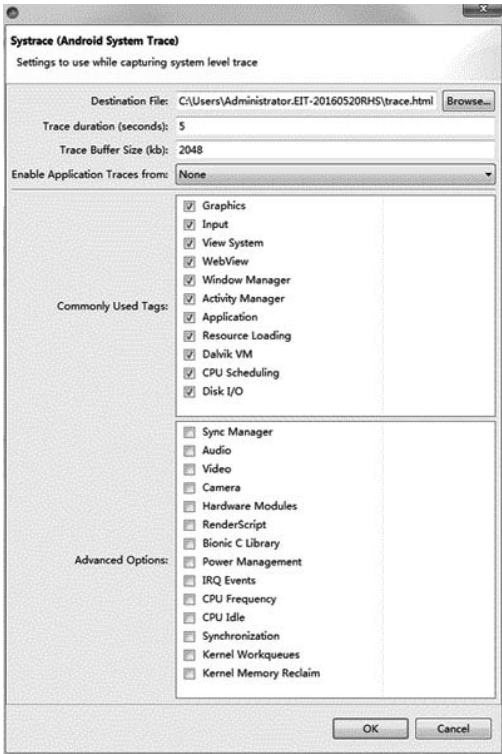


图 16-7 抓取设置界面

(3) 设置完成后，我们就来操作跟踪的过程。跟踪时间结束后，生成 trace.html 文件。

(4) 用 Chrome 打开 trace.html 文件进行分析。分析的方法后面会讲到。

2. 用命令行使用 Systrace

Android 提供一个 Python 脚本文件 systrace.py，它位于 Android SDK 目录/tools/systrace 中，我们可以执行以下命令来使用 Systrace：

```
$ cd android-sdk/platform-tools/systrace
$ python systrace.py --time=10 -o newtrace.html sched gfx view wm
```

3. 在代码中使用 Systrace

Systrace 并不会追踪应用的所有工作，在 Android 4.3 及以上版本的代码中，可以使用 Trace 类对应用中的具体活动进行追踪。Android 源码中也引用了 Trace 类，为了展示得更直观，这里 RecyclerView 的源码为 Android 6.0 版本的（Android 7.0 及以上版本展示不直观）：

```
frameworks/support/v7/recyclerview/src/android/support/v7/widget/RecyclerView.java

private final Runnable mUpdateChildViewsRunnable = new Runnable() {
    public void run() {
        if (!mFirstLayoutComplete) {
            return;
        }
        if (mDataSetHasChangedAfterLayout) {
            TraceCompat.beginSection(TRACE_ON_DATA_SET_CHANGE_LAYOUT_TAG);
            dispatchLayout();
            TraceCompat.endSection();
        } else if (mAdapterHelper.hasPendingUpdates()) {
            TraceCompat.beginSection(TRACE_HANDLE_ADAPTER_UPDATES_TAG);
            eatRequestLayout();
            mAdapterHelper.preProcess();
            if (!mLayoutRequestEaten) {
                rebindUpdatedViewHolders();
            }
            resumeRequestLayout(true);
            TraceCompat.endSection();
        }
    }
};
```

TraceCompat 类对 Trace 类进行了封装，其中 beginSection 方法和 endSection 方法之间的代码会被追踪，endSection 方法只会结束最近的 beginSection 方法，因此要保证 beginSection 方法和 endSection 方法的调用次数要相同。

16.1.3.2 用 Chrome 分析 Systrace

通过前面的方法生成的 trace.html 可以用 Chrome 打开，打开后的效果如图 16-8 所示。



图 16-8 用 Chrome 打开 trace.html

我们可以使用 W 键和 S 键进行放大和缩小，A 键和 D 键进行左右移动，接下来介绍图 16-8 中的各个视图区域。

1. Alert 区域

首先来看 Alert 区域，这一区域会标记出性能有问题的点，单击叹号图标就可以查看某一个 Alert 的问题描述，如图 16-9 所示。

1 item selected. Alert (1)	
Alert	Expensive measure/layout pass
Time spent	3.083 ms
measure	took 0.09ms
layout	took 3.00ms
Frame	
Description	Measure/Layout took a significant time, contributing to jank. Avoid triggering layout during animations.
Video Link	Android Performance Patterns: Invalidations, Layouts, and Performance

图 16-9 Alert 区域

这个 Alert 指出了 View 在 Measure/Layout 时耗费了大量的时间，导致出现 jank（同一帧画了多次）。给出的建议是避免在动画播放期间控制布局。

2. CPU 区域

接下来我们来查看 CPU 区域，每一行代表一个 CPU 核心和它执行任务的时间片，放大后会看到每个色块代表一个执行的进程，色块的长度代表其执行时间，如图 16-10 所示。



图 16-10 CPU 区域

图中 CPU 0 主要执行 adbb 线程和 InputReader 线程，CPU 1 主要执行了 surfaceflinger 线程和 ordinatorlayout 进程中的 RenderThread 线程，我们单击 RenderThread 色块，会给出 RenderThread 的相关信息，如图 16-11 所示。

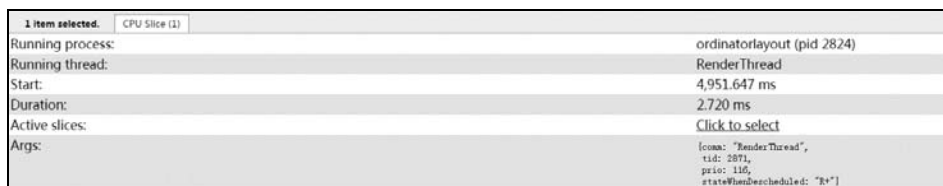


图 16-11 RenderThread 的相关信息

图 16-11 给出了当前色块所运行的线程和进程、开启时间和持续时间等信息。

3. 应用区域

应用区域会显示应用的帧数，如图 16-12 所示。

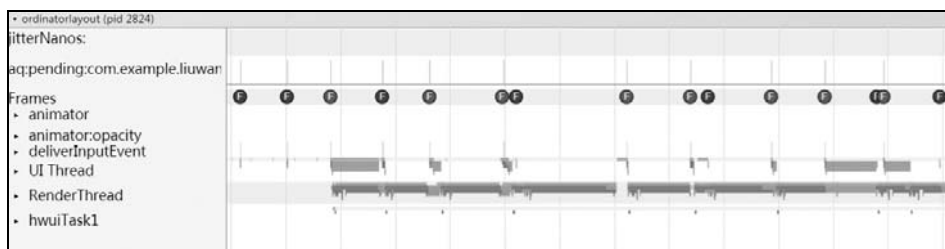


图 16-12 应用区域

Systrace 会给出应用中的 Frames 分析，每一帧就是一个 F 圆圈，F 圆圈有三种颜色，其中绿色表示 Frame 渲染流畅，黄色和红色则代表渲染时间超过了 16.6ms，其中红的更严重一些。我们单击红色 F 圆圈，会给出该 Frame 的信息，如图 16-13 所示。

从图 16-3 中可以看出, Frame 给出了问题提示: Scheduling delay (调度延迟), 当一帧绘制时间超过 19ms 会触发该提示, 更何况这一帧已经将近 40ms 了。导致这一问题产生的原因主要是线程在绘制时, 在很长一段时间都没有分配到 CPU 时间片, 因此无法继续进行绘制。按 M 键来高亮该时间段, 我们来查看 CPU 的情况, 如图 16-14 所示。

1 item selected.	Frame (1)
Alert	Scheduling delay
Running	34.752 ms
Not scheduled, but runnable	4.966 ms
Sleeping	1.083 ms
Frame	
Description	Work to produce this frame was descheduled for several milliseconds, contributing to jank. Ensure that code on the UI thread doesn't block on work being done on other threads, and that background threads (doing e.g. network or bitmap loading) are running at android.os.Process#THREAD_PRIORITY_BACKGROUND or lower so they are less likely to interrupt the UI thread. These background threads should show up with a priority number of 130 or higher in the scheduling section under the Kernel process.

图 16-13 Frames 分析



图 16-14 CPU 的情况

可以看出这个时间段中两个 CPU 都在满负荷运行。至于具体是什么让 CPU 繁忙，则需要使用 Traceview 来进行分析。

4. Alerts 总体分析

单击最右边的 Alerts 按钮会给出 Alert 的总体分析，如图 16-15 所示。

Alert type	Count
Scheduling delay	102
Expensive measure/layout pass	2

图 16-15 Alerts 总体分析

Alerts 会给出 Alert 类型，以及出现的次数。有了这些总体的分析，方便开发者对该时间段的绘制性能有一个大概了解，便于进行下一步分析。由于 Systrace 是以系统的角度返回一些信息的，只能为我们提供一个概览，它的深度是有限的，我们可以用它来进行粗略的检查，以便了解大概的情况，但是如果要分析更详细的，比如要找到是什么让 CPU 繁忙，某些方法的调用次数等，则还要借助另一个工具 Traceview。

16.1.4 Traceview

TraceView 是 Android SDK 中自带的数据采集和分析工具。一般来说，通过 TraceView 我们可以得到以下两种数据：

- 单次执行耗时的方法。
- 执行次数多的方法。

16.1.4.1 使用 Traceview

要分析 Traceview，则首先要得到一个 trace 文件，trace 文件的获取有两种方式，分别是在 DDMS 中使用和在代码中加入调试语句，下面分别对这两种方式进行介绍。

1. DDMS 中使用

- (1) 首先我们要打开 Android Studio 的 Tool 中的 Android Device Monitor，并连接手机。
- (2) 选择相应的进程，并单击 Start Method Profiling 按钮。
- (3) 对应用中需要监控的点进行操作。
- (4) 单击 Stop Method Profiling 按钮，会自动跳到 TraceView 视图。

2. 在代码中加入调试语句

如果开发过程中出现不好复现的问题，则需要在代码中添加 TraceView 监控语句，代码如下所示：

```
Debug.startMethodTracing();  
...  
Debug.stopMethodTracing();
```

在开始监控的地方调用 startMethodTracing 方法，在需要结束监控的地方调用 stopMethodTracing 方法。系统会在 SD 卡中生成 trace 文件，将 trace 文件导出并用 SDK 中的 Traceview 打开即可。当然不要忘了在 manifest 中加入 <uses-permission android:name="android.permission.WRITE_EXTERNAL_STORAGE"/> 权限。

16.1.4.2 分析 Traceview

为了分析 Traceview，我们来举一个简单的例子来生成 trace 文件，这里采用第二种方式：在代码中加入调试语句，代码如下所示：

```

public class CoordinatorLayoutActivity extends AppCompatActivity {
    private ViewPager mViewPager;
    private TabLayout mTabLayout;
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_tab_layout);
        Debug.startMethodTracing("test");//1
        initView();
    }
    private void initView() {
        try {
            Thread.sleep(1000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
    @Override
    protected void onStop() {
        super.onStop();
        Debug.stopMethodTracing();
    }
}

```

在注释 1 处调用了 startMethodTracing 方法开始监控，其中 test 是生成的 trace 文件的名称。在 initView 中我们特意调用 sleep 方法来做耗时操作。在 onStop 方法中我们调用了 stopMethodTracing 方法结束监控。这时会在 SD 卡根目录生成 test.trace 文件，我们将该文件导出到桌面，用 Traceview 来分析 test.trace 文件，在 cmd 中执行如图 16-16 所示的命令。

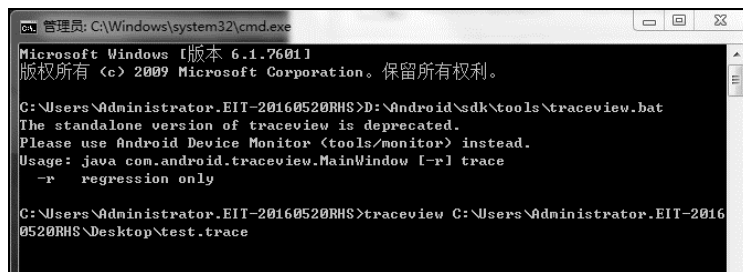


图 16-16 执行 traceview 命令

进入 traceview 所在的目录（直接将 traceview.bat 拖入到 cmd 中），并执行图 16-16 所示的 traceview 命令后会弹出 Traceview 视图，它分为两部分，分别是时间片面板和分析面

板，我们先来看时间片面板，如图 16-17 所示。

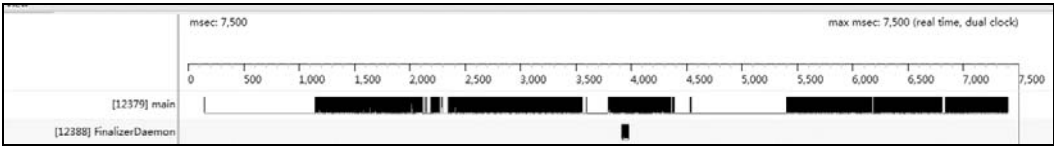


图 16-17 Traceview 视图的时间片面板

其中 X 轴代表时间的消耗，单位为 ms，Y 轴代表各个线程。一般会查看色块的长度，明显比较长的方法重点去关注，具体的分析还得看分析面板，如图 16-18 所示。

Name	Incl Cpu Time%	Incl Cpu Time	Excl Cpu Time%	Excl Cpu Time	Incl Real Time%	Incl Real Time	Excl Real Time%	Excl Real Time	Calls+Recur
0 android.os.Handler.dispatchMessage (Landroid/os/Me...	86.8%	3746.096	0.0%	1.154	44.3%	4553.182	0.0%	1.192	83+0
1 android.os.Handler.handleCallback (Landroid/os/Messa...	85.9%	3704.539	0.0%	1.124	43.8%	4504.067	0.0%	1.171	72+0
2 android.view.Choreographer\$FrameDisplayEventReceive...	82.8%	3569.476	0.0%	1.060	42.2%	4344.131	0.0%	1.049	68+0
3 android.view.Choreographer.doFrame (IJIV	82.7%	3568.416	0.2%	7.373	42.2%	4342.789	0.1%	7.725	68+0
4 android.view.Choreographer.doCallbacks (IJIV	82.3%	3549.034	0.3%	14.736	42.0%	4319.965	0.1%	15.373	270+0
5 android.view.Choreographer\$CallbackRecord.run (IJIV	81.6%	3521.523	0.1%	4.036	41.6%	4275.501	0.0%	4.069	185+0
6 android.view.ViewRootImpl\$TraverseRunnable.run (JV	63.6%	2744.458	0.0%	0.906	32.9%	3387.796	0.0%	0.895	60+0
7 android.view.ViewRootImpl.doTraverse (JV	63.6%	2743.552	0.1%	2.203	32.9%	3386.040	0.0%	2.236	60+0
8 android.view.ViewRootImpl.performTraversals (JV	63.4%	2736.564	0.2%	8.467	32.8%	3378.878	0.1%	8.774	60+0
9 android.support.v7.widget.LinearLayoutManager.fill (Lar...	29.8%	1286.573	0.0%	1.687	13.8%	1415.240	0.0%	1.704	32+0
10 android.support.v7.widget.LinearLayoutManager.layout...	29.2%	1260.939	0.1%	3.550	13.5%	1388.012	0.0%	3.594	36+0
11 android.view.ViewGroup.layout (IIIIIV	27.9%	1205.468	0.0%	1.395	13.0%	1333.297	0.0%	1.360	6+101
12 android.view.ViewGroup.layout (IIIIIV	27.9%	1205.387	0.1%	5.926	13.0%	1333.216	0.1%	6.046	6+185
13 android.widget.FrameLayout.onLayout (ZIIIIIV	27.8%	1199.104	0.0%	0.468	12.9%	1326.756	0.0%	0.473	6+36
14 android.widget.FrameLayout.layoutChildren (IIIIIZIV	27.8%	1199.035	0.1%	3.961	12.9%	1326.686	0.0%	4.023	6+38
15 android.widget.LinearLayout.onLayout (ZIIIIIV	27.6%	1192.050	0.0%	0.637	12.8%	1319.688	0.0%	0.623	4+46
16 android.view.ViewRootImpl.performTraverse (Landroid/v...	27.6%	1190.626	0.0%	0.104	12.8%	1317.080	0.0%	0.102	7+0

图 16-18 分析面板

每一列数据的代表的含义如表 16-1 所示。

表 16-1 分析面板列数据含义

列 名	含 义
Name	该线程运行过程中调用的函数名
Incl Cpu Time%	某个方法包括其内部调用的方法所占用 CPU 时间百分比
Excl Cpu Time%	某个方法不包括其内部调用的方法所占用 CPU 时间百分比
Incl Real Time%	某个方法包括其内部调用的方法所占用真实时间百分比
Excl Real Time%	某个方法不包括其内部调用的方法所占用真实时间百分比
Calls + Recur Calls / Total	某个方法次数+递归调用次数
Cpu Time / Call	该方法平均占用 CPU 时间
Cpu Time / Call	该方法平均占用真实时间
Incl Cpu Time	某个方法包括其内部调用的方法所占用 CPU 时间
Excl Cpu Time	某个方法不包括其内部调用的方法所占用 CPU 时间
Incl Real Time	某个方法包括其内部调用的方法所占用真实时间
Excl Real Time	某个方法不包括其内部调用的方法所占用真实时间

因为我们用 sleep 方法来进行耗时操作，所以可以单击 Incl Real Time 来进行降序排列，其中有很多系统调用的方法，我们一一进行过滤。最终我们发现了 CoordinatorLayoutActivity 的 initView 方法 Incl Real Time 的时间为 1000.493ms，这显然有问题，如图 16-19 所示。

Name	Incl Cpu...	Excl...	Incl Real Time	Excl Real Tim...	Excl Real Time	Calls+RecurC
2292 com.example.liuwangshu.mooncoordinatorlayout.CoordinatorLayoutActivity.initView (V)	0.0%	0.0%	1000.493	0.0%	0.049	1+
Parents (toplevel)	100.0%		1000.493			1/1
Children						
self	11.1%		0.049			
2228 java.lang.Thread.sleep (J/V)	88.9%		1000.444			1/2

图 16-19 分析面板

从图 16-19 中我们可以看出是调用 sleep 方法导致的耗时。关于 Traceview 还有很多种分析情况，就需要大家在平时进行积累了。

16.2 布局优化

一个界面的测量和绘制是通过递归来完成的，减少布局的层数就会减少测量和绘制的时间，从而性能就会得到提升。当然这只是布局优化的一方面，那么如何进行布局的分析和优化呢？本节内容会给你一个满意的答案。

16.2.1 布局优化工具

在讲到如何进行布局优化前，我们先来学习两种布局优化的工具，分别是 Hierarchy Viewer 和 Android Lint。

16.2.1.1 Hierarchy Viewer

Hierarchy Viewer 是 Android SDK 自带的可视化的调试工具，用来检查布局嵌套和绘制的时间。需要注意的是在 Android 的官方文档中提到：出于安全考虑，Hierarchy Viewer 只能连接 Android 开发版手机或模拟器。首先我们在 Android Studio 中选择 Tools→Android→Android Device Monitor，在 Android Device Monitor 中选择 Hierarchy Viewer，如图 16-20 所示。

选择 Hierarchy Viewer 后会打开 Hierarchy Viewer 窗口。

在 Hierarchy Viewer 窗口中有 4 个子窗口，它们的作用如下。

- Windows：当前设备所有界面列表。

- Tree View：将当前 Activity 的所有 View 的层次按照高层到低层从左到右显示出来。
- Tree Overview：全局概览，以缩略的形式显示。
- Layout View：整体布局图，以手机屏幕上真实的位置呈现出来。单击某一个控件，会在 Tree Overview 窗口中显示出对应的控件。

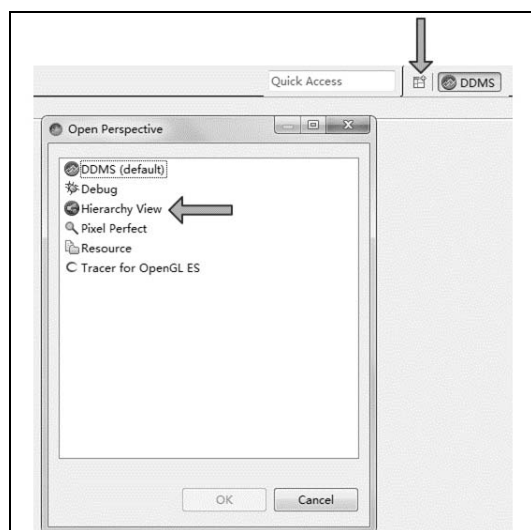


图 16-20 打开 Hierarchy Viewer

根据上面讲到的 Hierarchy Viewer 的 4 个子窗口，我们可以很容易地查看布局控件的层级关系。当然 Hierarchy Viewer 还可以查看某一个 View 的耗时，我们可以选择某一个 View，然后单击如图 16-21 所示的箭头标识的按钮，这里我们把他简称为 Layout Time 按钮。

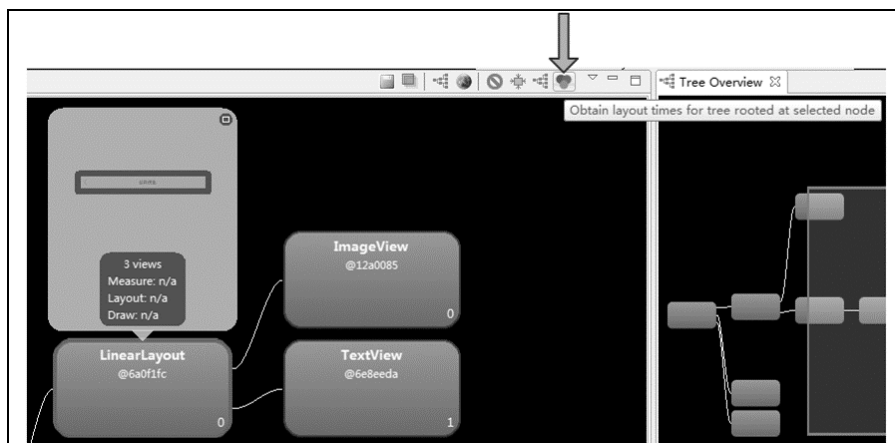


图 16-21 Layout Time 按钮

从图 16-21 中可以看出被选中的 RelativeLayout 自身的 Measure、Layout 和 Draw 的耗时数据都为 n/a。单击 Layout Time 按钮后, 就可以查看 View 的耗时情况了, 如图 16-22 所示。

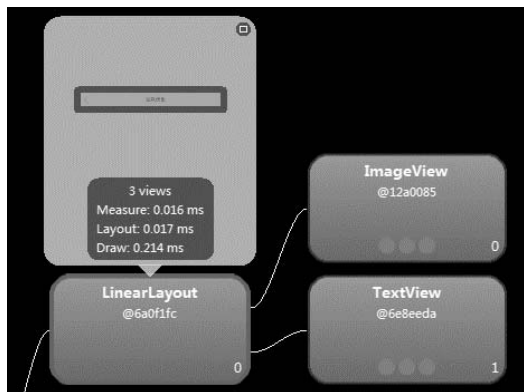


图 16-22 View 的耗时情况

被选中的 LinearLayout 给出了自身 Measure、Layout 和 Draw 的耗时, 并且它所包含的 View 中都有了 3 个指示灯, 分别代表当前 View 在 Measure、Layout 和 Draw 的耗时, 绿色代表比其他 50%View 的同阶段 (比如 Measure 阶段) 速度要快, 黄色代表比其他 50%View 同阶段速度要慢, 红色则代表比其他 View 同阶段都要慢, 出现红色就需要注意了。如果想要看 View 的具体耗时, 单击该 View 就可以了。

16.2.1.2 Android Lint

Android lint 是在 ADT 16 中提供的新工具, 它是一个代码扫描工具, 通过代码静态检查来发现代码出现的潜在问题, 并给出优化建议。检查的范围主要有以下几点:

- Correctness (正确性)。
- Security (安全性)。
- Performance (性能)。
- Usability (可用性)。
- Accessibility (可达性)。
- Internationalization (国际化)。

Android Lint 功能十分强大, 这里我们只关注 XML 布局检查, 可以通过 Android Studio 的 Analyze→Inspect Code 来配置检查的范围, 如图 16-23 所示。

单击图 16-24 中的 OK 按钮后, 就会进行代码检查, 检查的结果如图 16-24 所示。

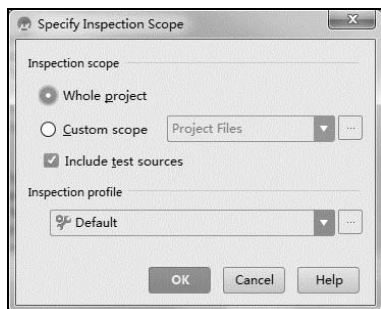


图 16-23 配置检查范围

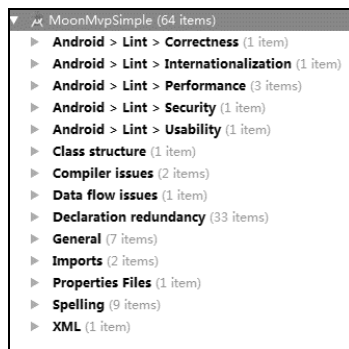


图 16-24 检查的结果

图 16-24 中列出了项目中出现的问题种类，以及每个问题种类的个数，问题种类包括我们前面提到的 Correctness、Internationalization、Performance 等。单击展开最后的 XML 一项，单击一个其中问题，就会出现如图 16-25 所示的提示。



图 16-25 XML 提示

图 16-26 给出了 Namespace declaration is never used 的提示，并指出了问题所在的文件和行数，我们单击数字 3，直接跳入到问题的代码，发现如下代码：

```
<?xml version="1.0" encoding="utf-8"?>
<RelativeLayout xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"//3
    android:id="@+id/activity_main"
    android:layout_width="match_parent"
    android:layout_height="match_parent">
    ...
```

注释 3 处的 Namespace 确实没有被用到。如果想要自定义 Android Lint 的检查提示，可以通过 File→Settings→Editor→Inspections 来配置 Android Lint，如图 16-26 所示。

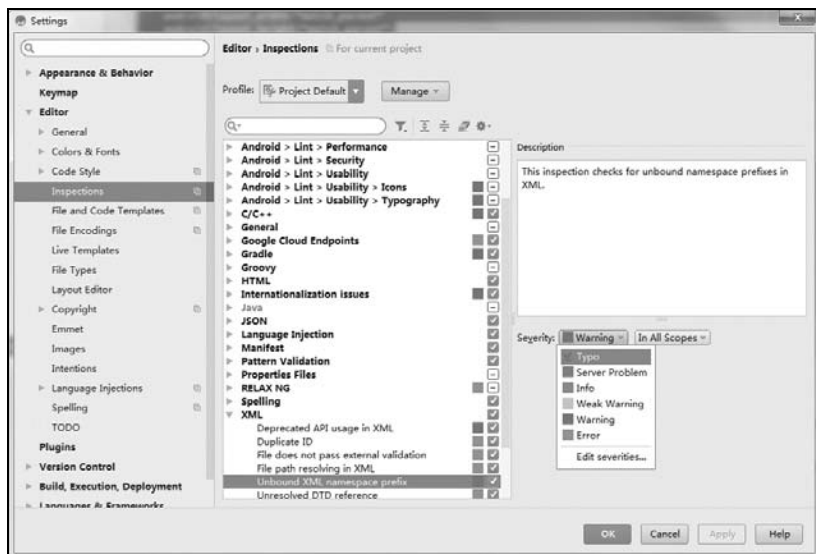


图 16-26 自定义 Android Lint 的检查提示

从图 16-27 可以发现，我们可以配置 Android Lint 检查的范围以及问题的严重等级。

16.2.2 布局优化方法

布局的优化方法有很多，主要包括合理运用布局、Include、Merge 和 ViewStub，下面对这些内容进行讲解。

16.2.2.1 合理运用布局

常用的布局主要有 LinearLayout、RelativeLayout 和 FrameLayout 等，合理地使用它们可以使得 Android 绘制工作量变少，性能得到提高，举个简单的例子：

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="horizontal">
    <TextView
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="布局优化" />
    <LinearLayout
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
```

```

        android:layout_marginLeft="10dp"
        android:orientation="vertical">
        <TextView
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:text="Merge" />
        <TextView
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:text="ViewStub" />
    </LinearLayout>
</LinearLayout>

```

上面的代码用了两个 LinearLayout 进行布局，运行效果如图 16-27 所示。



图 16-27 运行效果

我们用 Hierarchy Viewer 来查看层级情况，如图 16-28 所示。

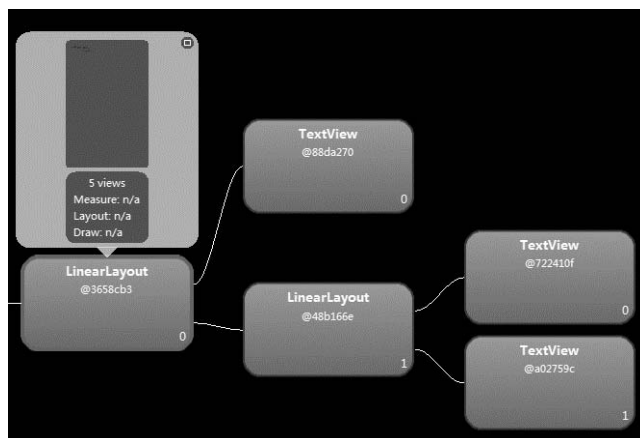


图 16-28 查看层级情况

可以看到我们的布局共有 3 层，一共含有 5 个 View。如果我们用 RelativeLayout 进行改写呢？代码如下所示：

```

<?xml version="1.0" encoding="utf-8"?>
<RelativeLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent">

```

```

<TextView
    android:id="@+id/tv_text1"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="布局优化" />
<TextView
    android:id="@+id/tv_text2"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_marginLeft="10dp"
    android:layout_toRightOf="@id/tv_text1"
    android:text="Merge" />
<TextView
    android:id="@+id/tv_text3"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_below="@id/tv_text2"
    android:layout_marginLeft="10dp"
    android:layout_toRightOf="@+id/tv_text1"
    android:text="ViewStub" />
</RelativeLayout>

```

我们只用了一个 RelativeLayout 进行布局,用 Hierarchy Viewer 查看层级情况,如图 16-29 所示。

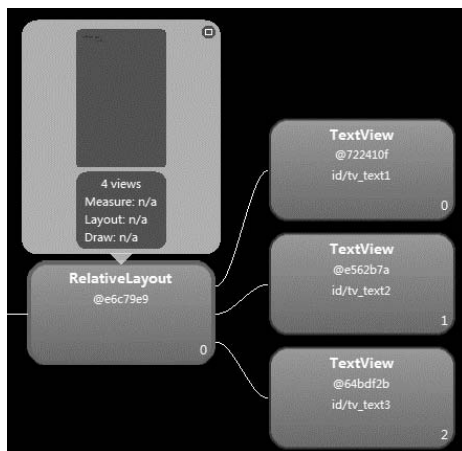


图 16-29 层级情况

布局共有两层,一共含有 4 个 View。从这里我们就可以看出用 RelativeLayout 减少了一层的布局,当然这只是一个简单例子,如果布局复杂,那么合理地用 RelativeLayout 来替代 LinearLayout 会减少很多层布局。如果根据上面的例子来看,RelativeLayout 的性能是

比 LinearLayout 低，因为 RelativeLayout 中的 View 的排列方式是基于彼此依赖的。但是在实际开发中面对的情况比较多，不能轻易地说谁的性能更好。在一般情况下，如果布局层数较多时，推荐用 RelativeLayout 来实现，如果布局嵌套较多，推荐使用 LinearLayout 来实现。

16.2.2.2 使用 Include 标签来进行布局复用

一个很常见的场景就是，多个布局需要复用一个相同的布局，比如一个 TitleBar。如果这些界面都要加上这个相同布局 TitleBar，维护起来就很麻烦，我们需要复制 TitleBar 的布局到每个需要添加的界面，这样容易发生遗漏。如果修改 TitleBar 则需要去每个引用 TitleBar 的布局中进行修改。为了解决这些问题，我们可以用 Include 标签来解决。

首先我们先来写一个简单的 TitleBar 布局：

titlebar.xml

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="40dp"
    android:background="@android:color/darker_gray">
    <ImageView
        android:layout_width="30dp"
        android:layout_height="30dp"
        android:src="@drawable/ico_left"
        android:padding="3dp"
        android:layout_gravity="center" />
    <TextView
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:layout_gravity="center"
        android:gravity="center"
        android:text="绘制优化" />
</LinearLayout>
```

这个 TitleBar 由 ImageView 和 TextView 组成，下面将 TitleBar 引入到我们此前用过的布局中，如下所示：

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="vertical">
    <include layout="@layout/titlebar" />
</RelativeLayout>
```

```

        android:layout_width="match_parent"
        android:layout_height="match_parent"
    >
    <TextView
        android:id="@+id/tv_text1"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="布局优化" />
    ...
</RelativeLayout>
</LinearLayout>

```

用 include 标签引入了 titlebar 布局，运行效果如图 16-30 所示。



图 16-30 运行效果

16.2.2.3 用 Merge 标签去除多余层级

Merge 意味着合并，在合适的场景使用 Merge 标签可以减少多余的层级。Merge 标签一般和 Include 标签搭配使用，上面的例子，我们用 Hierarchy Viewer 来查看布局层级，如图 16-31 所示。

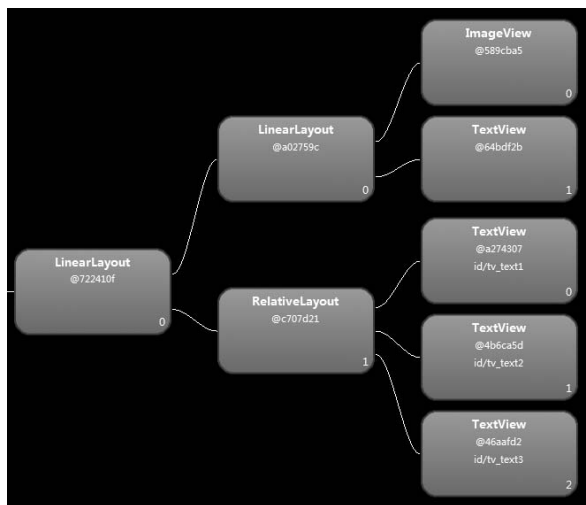


图 16-31 布局层级

Include 标签引用的布局的根布局是一个 LinearLayout。如果我们使用 Merge 标签来替换 LinearLayout 呢? titlebar.xml 的代码如下所示:

titlebar.xml

```
<?xml version="1.0" encoding="utf-8"?>
<merge xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="40dp"
    android:background="@android:color/darker_gray"
    >
    <ImageView
        android:layout_width="30dp"
        android:layout_height="30dp"
        android:src="@drawable/ico_left"
        android:padding="3dp"
        android:layout_gravity="center" />

    <TextView
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:layout_gravity="center"
        android:gravity="center"
        android:text="绘制优化" />
</merge>
```

这时我们再用 Hierarchy Viewer 来查看布局层级, 如图 16-32 所示。

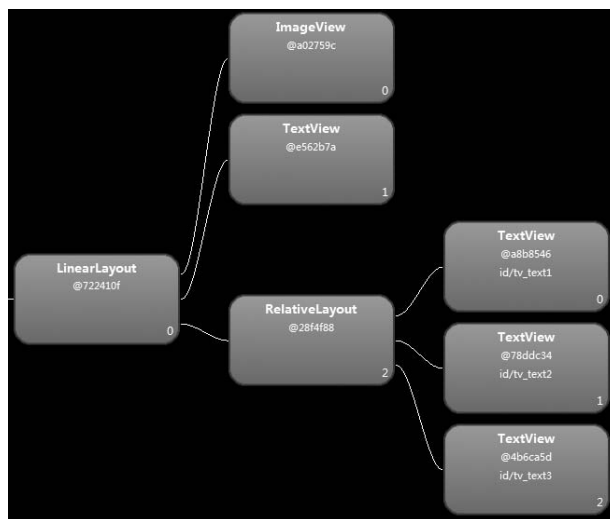


图 16-32 布局层级

此前的根布局 `LinearLayout` 没有了，但是这里用 `merge` 标签来替代 `LinearLayout` 会导致 `LinearLayout` 失效，布局就会错乱。`merge` 标签最好是替代 `FrameLayout` 或者布局方向一致的 `LinearLayout`，比如当前父布局 `LinearLayout` 的布局方向是垂直的，包含的子布局 `LinearLayout` 的布局方向也是垂直的，则可以用 `merge` 标签，而本场景 `TitleBar` 的根布局 `LinearLayout` 的布局方向是水平的，显然并不符合这一要求。但是如果执意想要在本场景中使用 `merge` 标签也是可以的，就是用继承自 `LinearLayout` 的自定义 `View`，代码如下所示：

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="vertical">
    <com.example.liuwangshu.moonlayout.TitleBar
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:orientation="horizontal"
        android:background="@android:color/darker_gray">
    </com.example.liuwangshu.moonlayout.TitleBar>
    <RelativeLayout
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        android:orientation="horizontal">
        <TextView
            android:id="@+id/tv_text1"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:text="布局优化" />
        ...
    </RelativeLayout>
</LinearLayout>
```

上面布局中 `TitleBar` 就是一个自定义 `View`，它继承自 `LinearLayout`，在 `TitleBar` 标签中添加此前的 `LinearLayout` 的属性：`android:orientation` 和 `android:background`，这样就达到了目的。

16.2.2.4 使用 `ViewStub` 来提高加载速度

一个很常见的开发场景就是我们想要一个布局时，并不是所有的控件都需要显示出来，而是显示出一部分，对于这种情况，我们一般采用的方法就是使用 `View` 的 `GONE` 和 `INVISIBLE` 属性，这种方法效率不高，虽然达到了隐藏的目的，但是仍在布局当中，系统仍然会解析它们，可以用 `ViewStub` 来解决这一问题。`ViewStub` 是轻量级的 `View`，不可见

并且不占布局位置。当 ViewStub 调用 inflate 方法或者设置可见时，系统会加载 ViewStub 指定的布局，然后将这个布局添加到 ViewStub 中，在对 ViewStub 调用 inflate 方法或者设置可见之前，它是不占布局空间和系统资源的，它主要的目的就是为目标视图占用一个位置。因此，使用 ViewStub 可以提高界面初始化的性能，从而提高界面的加载速度。首先在布局中加入 ViewStub 标签，布局代码如下所示：

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="vertical">
    <ViewStub
        android:id="@+id/viewsub"
        android:layout_width="match_parent"
        android:layout_height="40dp"
        android:layout="@layout/titlebar"/>
    ...
</LinearLayout>
```

在 ViewStub 标签中用 android:layout 引用了此前写好的布局 titlebar.xml。这时运行程序，ViewStub 标签所引用的布局是显示不出来的，因为该布局还没有加载到 ViewStub 中，接下来在代码中使用 ViewStub：

```
public class MainActivity extends AppCompatActivity {
    private ViewStub viewsub;
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        viewsub= (ViewStub) findViewById(R.id.viewsub);
        viewsub.inflate();//1
        viewsub.setVisibility(View.VISIBLE);//2
    }
}
```

注释 1 和注释 2 处的代码用来将 ViewStub 引用的布局加载到 ViewStub 中，这样引用的布局就显示出来了。在使用 ViewStub 时需要主要以下问题：

- ViewStub 只能加载一次，加载后 ViewStub 对象会被置为空，这样在 ViewStub 引用的布局被加载后，就不能用 ViewStub 来控制引用的布局了。因此，如果一个控件需要不断地显示和隐藏，还是要使用 View 的 Visibility 属性。
- ViewStub 不能嵌套 Merge 标签。

- ViewStub 操作的是布局文件，如果只是想操作具体的 View，还是要使用 View 的 Visibility 属性。

16.2.3 避免GPU过度绘制

什么是过度绘制呢？我们来打个比方，假设你要粉刷房子的墙壁，一开始刷了绿色，接着又刷了黄色，这样黄色就将绿色盖住，也就说明第一次的大量粉刷工作白做了。同样手机屏幕绘制也是如此，过度绘制是指在屏幕上某个像素在同一帧的时间内被绘制多次，从而浪费了 GPU 和 CPU 的资源。产生这一情况主要有两个原因：

- 在 XML 布局中，控件有重叠且都有设置背景。
- View 的 OnDraw 在同一区域绘制多次。

过度绘制是很难避免的，但是过多的过度绘制会浪费很多资源，并且导致性能问题，因此，避免过度绘制是十分必要的。我们可以用 Android 系统中自带的工具来检测过度绘制。首先要保证系统版本在 Android 4.1 以上，接着在开发者选项中打开调试 GPU 过度绘制选项就可以进入 GPU 过度绘制模式，如图 16-33 所示。



图 16-33 GPU 过度绘制模式

这时屏幕会出现出各种颜色，主要有以下几种。

- 白色：没有过度绘制——每个像素在屏幕上绘制了一次。
- 蓝色：一次过度绘制——每个像素点在屏幕上绘制了两次。
- 绿色：两次过度绘制——每个像素点在屏幕上绘制了三次。
- 粉色：三次过度绘制——每个像素点在屏幕上绘制了四次。
- 红色：四次或四次以上过度绘制——每个像素点在屏幕上绘制了五次或者五次以上。

合格的页面绘制是以白色和蓝色为主的，绿色以上区域不能超过整体的三分之一，颜色越浅越好。

避免过度绘制主要有以下两个方案：

- 移除不需要的 background。
- 在自定义 View 的 OnDraw 方法中，用 `canvas.clipRect` 来指定绘制的区域，防止重叠的组件发生过度绘制。

16.3 本章小结

本章围绕着绘制优化这一主题，介绍了绘制原理、绘制性能分析的工具和布局优化方法，这些内容能帮助开发者更高效地开发界面，预防过度绘制的产生。本章介绍了很多绘制优化相关的工具，很多工具的拓展使用知识都不少，这里只是带大家入个门，剩余的知识点还需要大家在实践中去进阶。

第 17 章

内存优化

关联章节：第 10 章 Java 虚拟机；第 11 章 Dalvik 和 ART；第 16 章 绘制优化

要学习 Android 的内存优化，首先要了解 Java 虚拟机和 DVM&ART，第 10 章和第 11 章就是给本章做铺垫的。本章从避免内存泄漏开始讲起，然后介绍常用的内存分析工具 Memory Monitor、Allocation Tracker 和 Heap Dump，最后介绍分析内存泄漏的利器 MAT 和 LeakCanary。

17.1 避免可控的内存泄漏

内存泄漏向来都是内存优化的重点，它如同幽灵一般存在于我们的应用当中，有时它不会现身，但一旦现身就会让你头疼不已。因此，如何避免、发现和解决内存泄漏就变得尤为重要。这一节我们先来学习如何避免内存泄漏。

17.1.1 什么是内存泄漏

每个应用程序都需要内存来完成工作，为了确保 Android 系统的每个应用都有足够的内存，Android 系统需要有效地管理内存分配。当内存不足时，Android 运行时就会触发 GC，GC 采用的垃圾标记算法为根搜索算法，具体内容请查看 10.6 节，如图 17-1 所示。

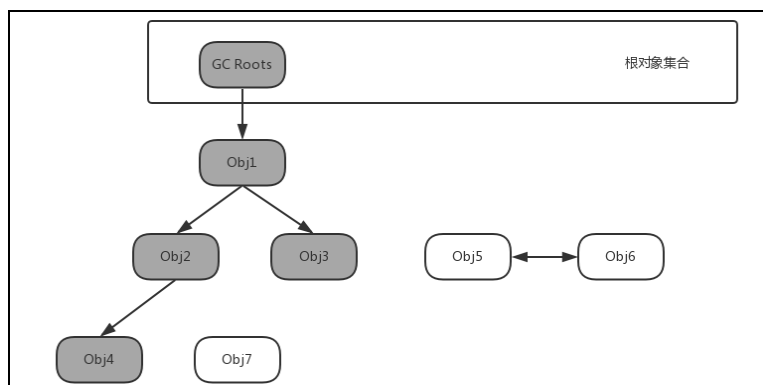


图 17-1 根搜索算法

从图 17-1 可以看出，Obj4 是可达的对象，表示它正被引用，因此不会标记为可回收的对象。Obj5、Obj6 和 Obj7 都是不可达的对象，其中 Obj5 和 Obj6 虽然互相引用，但是因为它们到 GC Roots 是不可达的，所以它们仍旧被标记为可回收的对象。

内存泄漏就是指没有用的对象到 GC Roots 是可达的（对象被引用），导致 GC 无法回收该对象。此时，如果 Obj4 是一个没有用的对象，但它仍与 GC Roots 是可达的，那么 Obj4 就会发生内存泄漏。内存泄漏产生的原因，主要分为三大类：

- 由开发人员自己编码造成的泄漏。
- 第三方框架造成的泄漏。
- 由 Android 系统或者第三方 ROM 造成的泄漏。

在通常情况下，第二种和第三种情况对于 Android 应用开发者来说是不可控的，但是第一种情况是可控的，既然是可控的，我们就要尽量在编码时避免造成内存泄漏，接下来列举出常见的内存泄漏的场景。

17.1.2 内存泄漏的场景

这里介绍一些内存泄漏的场景，当然这并不是全部，但是如果你都掌握了，基本也就够用了。

1. 非静态内部类的静态实例

非静态内部类会持有外部类实例的引用，如果非静态内部类的实例是静态的，就会间接地长期维持着外部类的引用，阻止被系统回收。代码如下所示：

```

public class SecondActivity extends AppCompatActivity {
    private static Object inner;
    private Button button;
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        button = (Button) findViewById(R.id.bt_next);
        button.setOnClickListener(new View.OnClickListener() {
            @Override
            public void onClick(View v) {
                createInnerClass();
                finish();
            }
        });
    }

    void createInnerClass() {
        class InnerClass {
        }
        inner = new InnerClass();//1
    }
}

```

当点击 Button 时，会在注释 1 处创建非静态内部类 InnerClass 的静态实例 inner，该实例的生命周期会和应用程序一样长，并且一直持有 SecondActivity 的引用，导致 SecondActivity 无法被回收。

2. 多线程相关的匿名内部类/非静态内部类

和前面的非静态内部类一样，匿名内部类也会持有外部类实例的引用。多线程相关的类有 AsyncTask 类、Thread 类和实现 Runnable 接口的类等，它们的匿名内部类/非静态内部类如果做耗时操作就可能发生内存泄漏，这里以 AsyncTask 的匿名内部类举例，如下所示：

```

public class AsyncTaskActivity extends AppCompatActivity {
    private Button button;
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_async_task);
        button = (Button) findViewById(R.id.bt_next);
        button.setOnClickListener(new View.OnClickListener() {
            @Override
            public void onClick(View v) {

```

```

        startAsyncTask();
        finish();
    }
});
}
void startAsyncTask() {
    new AsyncTask<Void, Void, Void>() { //1
        @Override
        protected Void doInBackground(Void... params) {
            while (true) ;
        }
    }.execute();
}
}

```

在注释 1 处实例化一个 AsyncTask，AsyncTask 的异步任务在后台执行耗时任务期间，AsyncTaskActivity 被销毁了，被 AsyncTask 持有的 AsyncTaskActivity 实例不会被垃圾收集器回收，直到异步任务结束。同理，自定义的 AsyncTask 如果是非静态内部类也会发生内存泄漏。解决办法就是自定义一个静态的 AsyncTask，如下所示：

```

public class AsyncTaskActivity extends AppCompatActivity {
    private Button button;
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_async_task);
        button = (Button) findViewById(R.id.bt_next);
        button.setOnClickListener(new View.OnClickListener() {
            @Override
            public void onClick(View v) {
                startAsyncTask();
                finish();
            }
        });
    }
    void startAsyncTask() {
        new MyAsyncTask().execute();
    }
    private static class MyAsyncTask extends AsyncTask<Void, Void, Void> {
        @Override
        protected Void doInBackground(Void... params) {
            while (true) ;
        }
    }
}

```

3. Handler 内存泄漏

Handler 的 Message 被存储在 MessageQueue 中，有些 Message 并不能马上被处理，它们在 MessageQueue 中存在的时间会很长，这就会导致 Handler 无法被回收。如果 Handler 是非静态的，则 Handler 也会导致引用它的 Activity 或者 Service 不能被回收。

```
public class HandlerActivity extends AppCompatActivity {
    private Button button;
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_handler);
        button = (Button) findViewById(R.id.bt_next);
        final Handler mHandler = new Handler() {
            @Override
            public void handleMessage(Message msg) {
                super.handleMessage(msg);
            }
        };
        button.setOnClickListener(new View.OnClickListener() {
            @Override
            public void onClick(View v) {
                mHandler.sendMessageDelayed(Message.obtain(), 60000);
                finish();
            }
        });
    }
}
```

Handler 是非静态的匿名内部类的实例，它会隐性引用外部类 HandlerActivity。上面的例子就是当我们点击 Button 时，HandlerActivity 会结束，但是 Handler 中的消息还没有被处理，因此 HandlerActivity 无法被回收。解决方案有两个，一个是使用一个静态的 Handler 内部类，Handler 持有的对象要使用弱引用：

```
public class HandlerActivity extends AppCompatActivity {
    private Button button;
    private MyHandler myHandler = new MyHandler(this);
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_handler);
        button = (Button) findViewById(R.id.bt_next);
        button.setOnClickListener(new View.OnClickListener() {
            @Override
```

```

        public void onClick(View v) {
            myHandler.sendMessageDelayed(Message.obtain(), 60000);
            finish();
        }
    });
}
public void show() {

}
private static class MyHandler extends Handler {
    private final WeakReference<HandlerActivity> mActivity;

    public MyHandler(HandlerActivity activity) {
        mActivity = new WeakReference<HandlerActivity2>(activity);
    }
    @Override
    public void handleMessage(Message msg) {
        if (mActivity != null && mActivity.get() == null) {
            mActivity.get().show();
        }
    }
}
}

```

MyHandler是一个静态的内部类，它持有的HandlerActivity对象使用了弱引用，这样就避免了内存泄漏。如果觉得麻烦，也可以使用避免内存泄漏的Handler开源库WeakHandler，地址为<https://github.com/badoo/android-weak-handler>。还有一个解决方案是在Activity的Destroy方法中移除MessageQueue中的消息，如下所示：

```

@Override
public void onDestroy() {
    if (myHandler != null) {
        myHandler.removeCallbacksAndMessages(null);
    }
    super.onDestroy();
}

```

在 onDestroy 方法中将 Callbacks 和 Messages 全部清除掉。采用这种解决方案的话，Handler 中的消息可能无法全部处理完，因此这里建议使用第一种解决方案。

4. 未正确使用 Context

对于不是必须使用 Activity 的 Context 的情况（Dialog 的 Context 必须使用 Activity 的

Context)，我们可以考虑使用 Application Context 来代替 Activity 的 Context，这样可以避免 Activity 泄漏，比如如下的单例模式：

```
public class AppSettings {
    private Context mAppContext;
    private static AppSettings mAppSettings = new AppSettings();
    public static AppSettings getInstance() {
        return mAppSettings;
    }

    public final void setup(Context context) {
        mAppContext = context;
    }
}
```

mAppSettings 作为静态对象，其生命周期会长于 Activity。当进行屏幕旋转时，在默认情况下，系统会销毁当前 Activity。因为当前 Activity 调用了 setup 方法，并传入了 Activity Context，使得 Activity 被一个单例持有，导致垃圾收集器无法回收，进而产生了内存泄漏。解决方法就是使用 Application 的 Context：

```
public final void setup(Context context) {
    mAppContext = context.getApplicationContext();
}
```

5. 静态 View

使用静态 View 可以避免每次启动 Activity 都去读取并渲染 View，但是静态 View 会持有 Activity 的引用，导致 Activity 无法被回收，解决的办法就是在 onDestroy 方法中将静态 View 置为 null，代码如下所示：

```
public class SecondActivity extends AppCompatActivity {
    private static Button button;
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        button = (Button) findViewById(R.id.bt_next);
        button.setOnClickListener(new View.OnClickListener() {
            @Override
            public void onClick(View v) {
                finish();
            }
        });
    }
}
```

6. WebView

不同的 Android 版本的 WebView 会有差异，加上不同厂商定制 ROM 的 WebView 的差异，这就导致 WebView 存在着很大的兼容性问题。WebView 都会存在内存泄漏的问题，在应用中只要使用一次 WebView，内存就不会被释放掉。通常的解决办法就是为 WebView 单开一个进程，使用 AIDL 与应用的主进程进行通信。WebView 进程可以根据业务需求，在合适的时机进行销毁。

7. 资源对象未关闭

资源对象比如 Cursor、File 等，往往都使用了缓冲，会造成内存泄漏。因此，在资源对象不使用时，一定要确保它们已经关闭并将它们的引用置为 null，通常在 finally 语句中进行关闭，防止出现异常时，资源未被释放的问题。

8. 集合中对象没清理

通常把一些对象的引用加入到了集合中，当不需要该对象时，如果没有把它的引用从集合中清理掉，这样这个集合就会越来越大。如果这个集合是 static 的话，那情况就会更加严重。

9. Bitmap 对象

临时创建的某个相对比较大的 Bitmap 对象，在经过变换得到新的 Bitmap 对象之后，应该尽快回收原始的 Bitmap，这样能够更快释放原始 Bitmap 所占用的空间。避免静态变量持有比较大的 Bitmap 对象或者其他大的数据对象，如果已经持有，要尽快置空该静态变量。

10. 监听器未关闭

很多系统服务（比如 TelephonyMannager、SensorManager）需要 register 和 unregister 监听器，我们需要确保在合适的时候及时 unregister 那些监听器。自己手动添加的 Listener，要记得在合适的时候及时移除这个 Listener。

17.2 Memory Monitor

要想做好内存优化工作，就要掌握两大部分的知识，一部分是理解内存优化相关的原理，另一部分就是善于运用内存分析的工具，本节就来讲解 Memory Monitor。在 Android Studio(以下简称 AS)中 Android Monitor 是一个主窗口，它包含了 Logcat、Memory Monitor、

CPU Monitor、GPU Monitor 和 Network Monitor。其中 Memory Monitor 可以轻松地监视应用程序的性能和内存使用情况，以便于找到被分配的对象，定位内存泄漏，并跟踪连接设备中正在使用的内存数量。Memory Monitor 可以报告出你的应用程序的内存分配情况，更形象地呈现出应用程序使用的内存。它的作用如下：

- 实时显示可用的和分配的 Java 内存的图表。
- 实时显示垃圾收集（GC）事件。
- 启动垃圾收集事件。
- 快速测试应用程序的缓慢是否与过度的垃圾收集事件有关。
- 快速测试应用程序崩溃是否与内存耗尽有关。

17.2.1 使用Memory Monitor

在使用 Memory Monitor 之前要确保手机开启了开发者模式和 USB 调试，使用的步骤如下：

- (1) 运行需要监控的应用程序。
- (2) 单击 AS 面板下面的 Android 图标，并选择 Monitors 选项。

如果 Memory Monitor 已经运行，效果如图 17-2 所示。

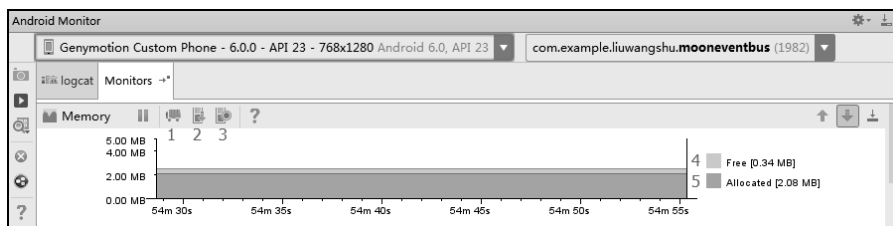


图 17-2 Memory Monitor

图中的标注的功能如下。

- Initiate GC（标识 1）：用来手动触发 GC。
- Dump Java heap（标识 2）：保存内存快照。
- Start/Stop Allocation Tracking（标识 3）：打开 Allocation Tracker 工具（17.3 节会介绍）。
- Free（标识 4）：当前应用未分配的内存大小。
- Allocated（标识 5）：当前应用分配的内存大小。

图 17-2 中 Y 轴显示当前应用的分配的内存和未分配的内存大小；X 轴表示经过的时间。

17.2.2 大内存申请与GC

从图 17-3 可以看出，分配的内存急剧上升，这就是大内存分配的场景，我们要判断这是否是合理的分配的内存，是 Bitmap 还是其他的大数据，并且对这种大数据进行优化，减少内存开销。接下来分配的内存出现急剧下降，这表示垃圾收集事件，用来释放内存。

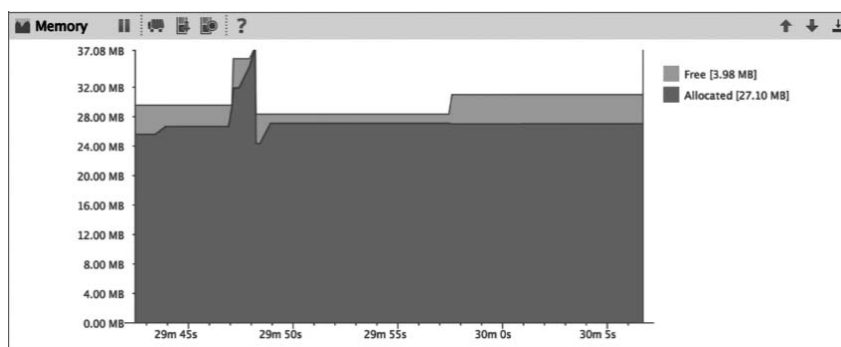


图 17-3 大内存申请与 GC

17.2.3 内存抖动

内存抖动一般指在很短的时间内发生了多次内存分配和释放，严重的内存抖动还会导致应用程序卡顿。内存抖动出现的原因主要是短时间频繁地创建对象（可能在循环中创建对象），内存为了应对这种情况，也会频繁地进行 GC。非并行 GC 在进行时，其他线程都会被挂起，等待 GC 操作完成后恢复工作。如果是频繁的 GC 就会产生大量的暂停时间，这会导致界面绘制时间减少，从而使得多次绘制一帧的时长超过了 16ms，产生的现象就是界面卡顿。综合起来就产生了内存抖动，产生了锯齿状的抖动图示，如图 17-4 所示。

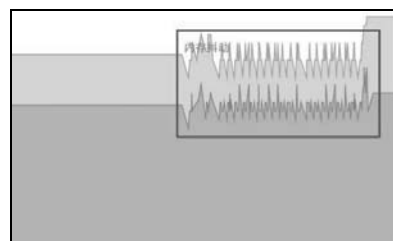


图 17-4 内存抖动

17.3 Allocation Tracker

Allocation Tracker 用来跟踪内存分配，它允许你在执行某些操作的同时监视在何处分配对象，了解这些分配使你能够调整与这些操作相关的方法调用，以优化应用程序性能和内存使用。Allocation Tracker 能够做到如下的事情：

- 显示代码分配对象类型、大小、分配线程、堆栈跟踪的时间和位置。
- 通过重复的分配/释放模式帮助识别内存变化。
- 当与 HPROF Viewer 结合使用时，可以帮助你跟踪内存泄漏。例如，如果你在堆上看到一个 Bitmap 对象，你可以使用 Allocation Tracker 来找到其分配的位置。

17.3.1 使用Allocation Tracker

AS 和 DDMS 中都有 Allocation Tracker, 这里只介绍 AS 中的 Allocation Tracker 如何使用。首先要确保手机开启了开发者模式，并且开启了 USB 调试。使用的步骤如下：

- (1) 运行需要监控的应用程序。
- (2) 单击 AS 面板下面的 Android 图标，并选择 Monitors 选项。
- (3) 单击 Start Allocation Tracking 按钮，这时 Start Allocation Tracking 按钮变为了 Stop Allocation Tracking 按钮。
- (4) 操作应用程序。
- (5) 单击 Stop Allocation Tracking 按钮，结束快照。这时 Memory Monitor 会显示出捕获快照的期间，如图 17-5 所示。

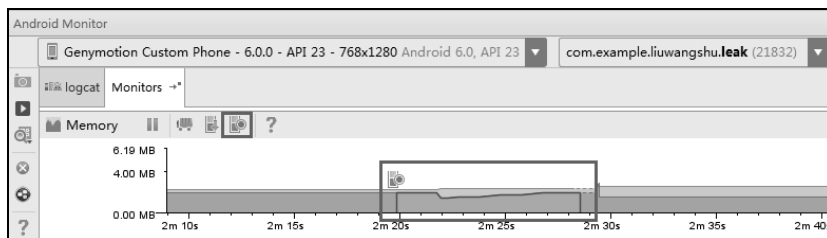
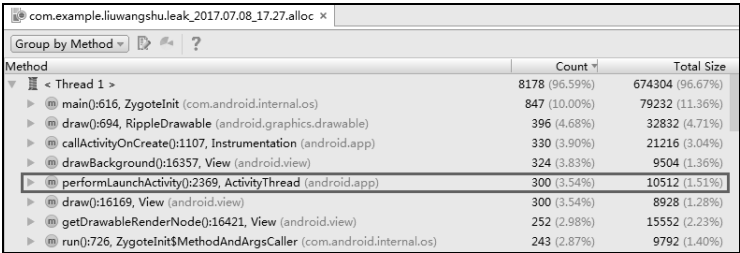


图 17-5 捕获快照的期间

- (6) 过几秒后就会自动打开一个窗口，显示当前生成的 alloc 文件的内存数据。

17.3.2 alloc文件分析

自动打开的 alloc 文件窗口如图 17-6 所示。



Method	Count	Total Size
< Thread 1 >	8178 (96.59%)	674304 (96.67%)
main():616, ZygoteInit (com.android.internal.os)	847 (10.00%)	79232 (11.36%)
draw():694, RippleDrawable (android.graphics.drawable)	396 (4.68%)	32832 (4.71%)
callActivityOnCreate():1107, Instrumentation (android.app)	330 (3.90%)	21216 (3.04%)
drawBackground():16357, View (android.view)	324 (3.83%)	9504 (1.36%)
performLaunchActivity():2369, ActivityThread (android.app)	300 (3.54%)	10512 (1.51%)
draw():16169, View (android.view)	300 (3.54%)	8928 (1.28%)
getDrawableRenderNode():16421, View (android.view)	252 (2.98%)	15552 (2.23%)
run():726, ZygoteInit\$MethodAndArgsCaller (com.android.internal.os)	243 (2.87%)	9792 (1.40%)

图 17-6 alloc 文件窗口

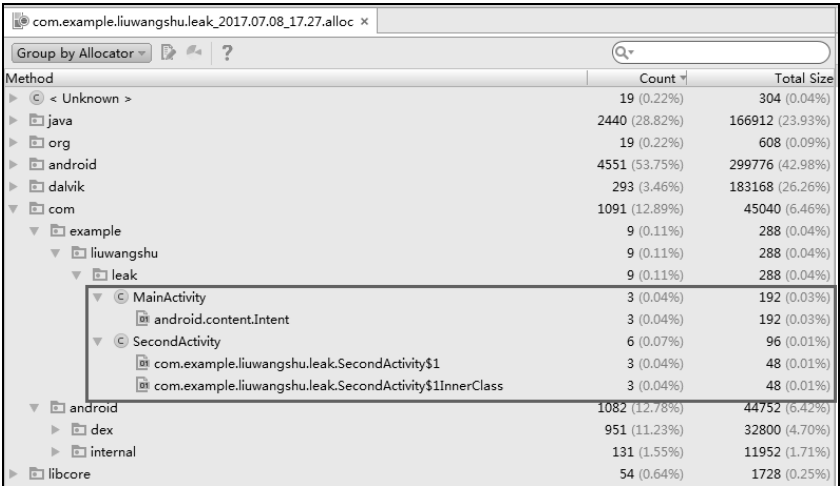
该 alloc 文件窗口列出如表 17-1 所示的信息。

表 17-1 alloc 文件窗口列出的信息

列	说 明
Method	负责分配的 Java 方法
Count	分配的实例总数
Total Size	分配内存的总字节数

接着我们分析图 17-6 中加方框的内容，负责分配的 Java 方法为 performLaunchActivity，内存分配序列为 2369，分配的对象为 ActivityThread，分配的实例总数为 300 个，分配内存的总字节数为 10512。不了解 performLaunchActivity 方法和 ActivityThread 的读者可以参考第 4 章。

目前的列表选项是 Group by Method，我们也可以选择 Group By Allocator，如图 17-7 所示。



Method	Count	Total Size
< Unknown >	19 (0.22%)	304 (0.04%)
java	2440 (28.82%)	166912 (23.93%)
org	19 (0.22%)	608 (0.09%)
android	4551 (53.75%)	299776 (42.98%)
dalvik	293 (3.46%)	183168 (26.26%)
com	1091 (12.89%)	45040 (6.46%)
example	9 (0.11%)	288 (0.04%)
liuwangshu	9 (0.11%)	288 (0.04%)
leak	9 (0.11%)	288 (0.04%)
MainActivity	3 (0.04%)	192 (0.03%)
android.content.Intent	3 (0.04%)	192 (0.03%)
SecondActivity	6 (0.07%)	96 (0.01%)
com.example.liuwangshu.leak.SecondActivity\$1	3 (0.04%)	48 (0.01%)
com.example.liuwangshu.leak.SecondActivity\$1InnerClass	3 (0.04%)	48 (0.01%)
android	1082 (12.78%)	44752 (6.42%)
dex	951 (11.23%)	32800 (4.70%)
internal	131 (1.55%)	11952 (1.71%)
libcore	54 (0.64%)	1728 (0.25%)

图 17-7 Group By Allocator 的信息

为了更好地解释图 17-7 中的信息,这里给出测试的代码,MainActivity 和 SecondActivity 的代码如下所示:

MainActivity.java

```
public class MainActivity extends AppCompatActivity {
    private Button button;
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        button = (Button)findViewById(R.id.bt_next);
        button.setOnClickListener(new View.OnClickListener() {
            @Override
            public void onClick(View v) {
                startActivity(new Intent(MainActivity.this,SecondActivity.class));
            }
        });
    }
}
```

SecondActivity.java

```
public class SecondActivity extends AppCompatActivity {
    private static Object inner;
    private Button button;
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        button = (Button) findViewById(R.id.bt_next);
        button.setOnClickListener(new View.OnClickListener() {
            @Override
            public void onClick(View v) {
                createInnerClass();
                finish();
            }
        });
    }
    void createInnerClass() {
        class InnerClass {
        }
        inner = new InnerClass();
    }
}
```

其中 SecondActivity 是存在内存泄漏的,生成快照期间,我的操作就是在 MainActivity 和 SecondActivity 之间跳转了 3 次(点击 Button 共 6 次)。这时我们回过头来看图 17-7 方框中的信息,MainActivity 总共分配了 3 个 Intent 实例,占用内存为 192 字节。SecondActivity 总共分配了 6 个实例,占用内存为 96 字节,其中分配了 3 个匿名内部类 OnClickListener 的实例,3 个 InnerClass 的实例。

我们可以选择列表中的一项,单击鼠标右键,在弹出的菜单中选择 jump to the source 就可以跳转到对应的源文件中。除此之外,还可以单击 Show/Hide Chart 按钮来显示数据的图形化,如图 17-8 所示。

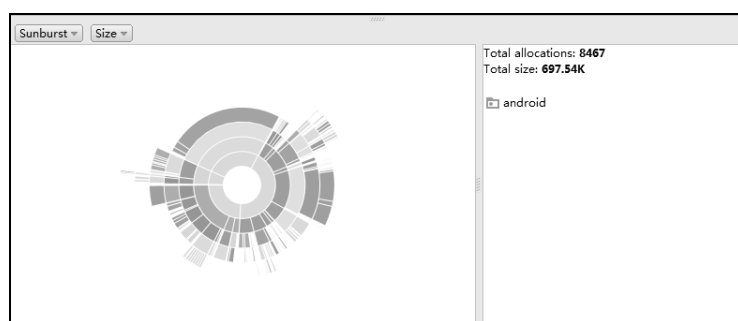


图 17-8 数据的图形化

17.4 Heap Dump

Heap Dump 的主要功能就是查看不同的数据类型在内存中的使用情况。它可以帮助你找到大对象,也可以通过数据的变化发现内存泄漏。

17.4.1 使用Heap Dump

打开 Android Device Monitor 工具,在左边 Devices 列表中选择要查看的应用程序进程,单击 Update Heap 按钮(一半是绿色的圆柱体),在右边选择 Heap 选项,并单击 Cause GC 按钮,就开始显示数据。我们每次单击 Cause GC 按钮都会强制应用程序进行垃圾回收,并将清理后的数据显示在 Heap 工具中,如图 17-9 所示。

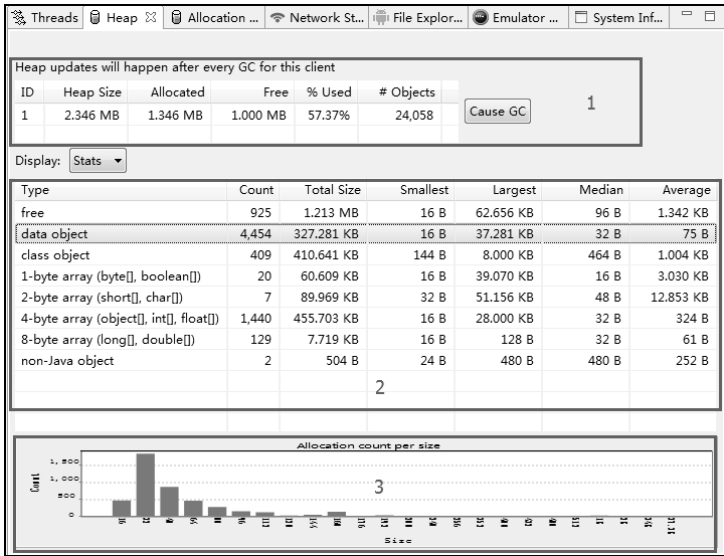


图 17-9 Heap Dump 视图

Heap Dump 共有三个区域，分别是总览视图（标识 1）、详情视图（标识 2）和内存分配柱状图（标识 3）。

1. 总览视图

其中总览视图可以查看整体的内存情况，其所列内容的说明如表 17-2 所示。

表 17-2 总览视图所列内容的说明

列	说 明
Heap Size	堆栈分配给该应用程序的内存大小
Allocated	已分配使用的内存大小
Free	空闲的内存大小
%Used	当前 Heap 的使用率（Allocated/Heap Size）
#Objects	对象的数量

结合表 17-2 和图 17-9，我们在总览视图获得的信息就是：堆栈分配给当前的应用程序的内存大小为 2.346MB，已分配的内存为 1.346MB，空闲的内存为 1MB，当前 Heap 的使用率为 57.37%，对象的数量为 24058 个。

2. 详情视图

详细视图展示了所有的数据类型的内存情况，其列的含义如表 17-3 所示。

表 17-3 详细视图列的含义

列	说 明
Type	数据类型
Total Size	总共占用的内存大小
Smallest	将该数据类型的对象从小到大排列，排在第一个的对象所占用的内存
Largest	将该数据类型的对象从小到大排列，排在最后一个的对象所占用的内存
Median	将该数据类型的对象从小到大排列，排在中间的对象所占用的内存
Average	该数据类型的对象所占用内存的平均值

除了列的含义，还有行的含义，如表 17-4 所示。

表 17-4 详细视图行的含义

行	说 明
free	内存碎片
data object	对象
class object	类
1-byte array (byte[],boolean[])	1 字节的数组对象
2-byte array (short[],char[])	2 字节的数组对象
4-byte array (object[],int[],float[])	4 字节的数组对象
8-byte array (long[],double[])	8 字节的数组对象
non-Java object	非 Java 对象

比较重要的是 free 这一行的信息，它与总览视图中的 free 的含义不同，它代表内存碎片。当新创建一个对象时，如果碎片内存能容下该对象，则复用碎片内存，否则就会从 free 空间（总览视图中的 free）重新划分内存给这个新对象。free 是判断内存碎片化程度的一个重要的指标。此外，1-byte array 这一行的信息也很重要，因为图片是以 byte[] 的形式存储在内存中的，如果 1-byte array 一行的数据过大，则需要检查图片的内存管理了。

17.4.2 检测内存泄漏

Heap Dump 也可以检测内存泄漏。在左边 Devices 列表中选择要查看的应用程序进程，单击 Update Heap 按钮（一半是绿色的圆柱体），在右边选择 Heap 选项，并单击 Cause GC 按钮，就开始显示数据，如图 17-10 所示。

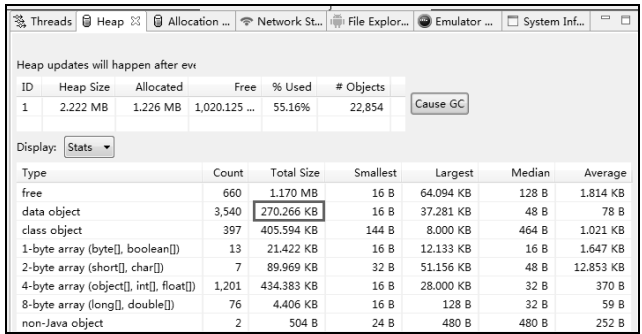


图 17-10 检测内存泄漏

这时 data object 的 Total Size 为 270.266KB。接下来操作应用,这个应用仍旧是在 17.3.2 节所举的内存泄漏的例子,反复地在 MainActivity 和 SecondActivity 之间跳转了 10 次 (单击 Button 共 20 次),数据显示如图 17-11 所示。

Type	Count	Total Size	Smallest	Largest	Median	Average
free	1,620	1.079 MB	16 B	66.062 KB	64 B	698 B
data object	12,101	768.172 KB	16 B	37.281 KB	32 B	65 B
class object	409	410.641 KB	144 B	8.000 KB	464 B	1.004 KB

图 17-11 单击 Button 共 20 次的的数据

data object 的 Total Size 变为了 768.172KB。这时单击 Cause GC 按钮,数据显示如图 17-12 所示。

Type	Count	Total Size	Smallest	Largest	Median	Average
free	1,446	1.530 MB	16 B	76.062 KB	128 B	1.083 KB
data object	6,459	444.516 KB	16 B	37.281 KB	32 B	70 B
class object	409	410.641 KB	144 B	8.000 KB	464 B	1.004 KB

图 17-12 单击 Cause GC 按钮的数据

可以看到 data object 的 Total Size 变为了 444.516KB,再单击一次 Cause GC 按钮,数据显示如图 17-13 所示。

Type	Count	Total Size	Smallest	Largest	Median	Average
free	942	1.697 MB	16 B	104.312 KB	160 B	1.845 KB
data object	4,392	323.312 KB	16 B	37.281 KB	32 B	75 B
class object	409	410.641 KB	144 B	8.000 KB	464 B	1.004 KB

图 17-13 再次单击 Cause GC 按钮的数据

Total Size 变为了 323.312KB,经过两次 Cause GC 的操作, Total Size 的值从 768.172KB 变为了 323.312KB,这是一个比较大的变化,说明在 Cause GC 操作之前有 462.86KB (768.172KB-323.312KB) 的内存没有被回收,可能发生了内存泄漏。

17.5 内存分析工具MAT

在进行内存分析时，我们可以使用Memory Monitor和Heap Dump观察内存的使用情况，使用Allocation Tracker跟踪内存分配的情况，也可以通过这些工具来找到疑似发生内存泄漏的位置。但是如果想要深入地进行分析并确定内存泄漏，就要分析疑似发生内存泄漏时所生成的堆存储文件。堆存储文件可以使用DDMS或者Memory Monitor来生成，输出的文件格式为hprof，而MAT就是分析堆存储文件的。MAT全称为Memory Analysis Tool，是对内存进行详细分析的工具，它是Eclipse的插件，如果用Android Studio进行开发则需要单独下载它，下载地址为<http://eclipse.org/mat>，这里讲解的MAT的版本为 1.6.1。

17.5.1 生成hprof文件

MAT 用来分析 hprof 文件，首先我们要学习如何生成 hprof 文件，主要有两种方式，分别是 DDMS 生成 hprof 文件和 Memory Monitor 生成 hprof 文件。

1. 准备内存泄漏的代码

我们需要准备一段发生内存泄漏的代码，如下所示：

```
public class MainActivity extends AppCompatActivity {
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        LeakThread leakThread = new LeakThread();
        leakThread.start();
    }
    class LeakThread extends Thread {
        @Override
        public void run() {
            try {
                Thread.sleep(60 * 60 * 1000);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
}
```

上面的代码是很典型的内存泄漏的例子，原因就是非静态内部类 LeakThread 持有外部类 MainActivity 的引用，LeakThread 中做了耗时操作，导致 MainActivity 无法被释放。

2. DDMS 生成 hprof 文件

生成 hprof 文件主要分为以下几个步骤：

- (1) 在 Android Studio 中打开 DDMS，运行程序。
- (2) 在 Devices 中选择要分析的应用程序进程，单击 Update Heap 按钮（一半是绿色的圆柱体）开始进行追踪。
- (3) 进行可能发生内存问题的操作（本文的例子就是不断地切换横竖屏）。
- (4) 单击 Dump HPROF File 按钮结束追踪，生成并保存 hprof 文件，如图 17-14 所示。

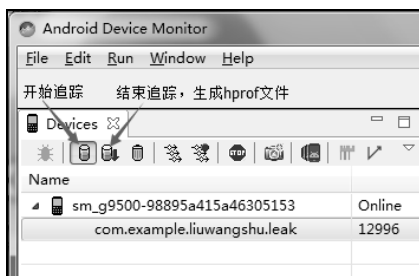


图 17-14 DDMS 生成 hprof 文件

DDMS 生成的 hprof 文件并不是标准的，还需要将它转换为标准的 hprof 文件，这样才会被 MAT 识别从而进行分析，可以使用 SDK 自带的 hprof-conv 进行转换，它的路径在 sdk/platform-tools 中，进入到该路径执行以下语句即可：

```
hprof-conv D:\before.hprof D:\after.hprof
```

其中 D:\before.hprof 是要转换的 hprof 文件路径，D:\after.hprof 则是转换后 hprof 文件的保存路径。

3. Memory Monitor 生成 hprof 文件

除了用 DDMS 来生成 hprof 文件外，还可以用 AS 的 Memory Monitor 来生成 hprof 文件。生成 hprof 文件主要分为以下几个步骤：

- (1) 在 Android Monitor 中选择要分析的应用程序进程。
- (2) 进行可能发生内存问题的操作（本文的例子就是不断地切换横竖屏）。
- (3) 单击 Dump Java Heap 按钮，生成 hprof 文件，如图 17-15 所示。

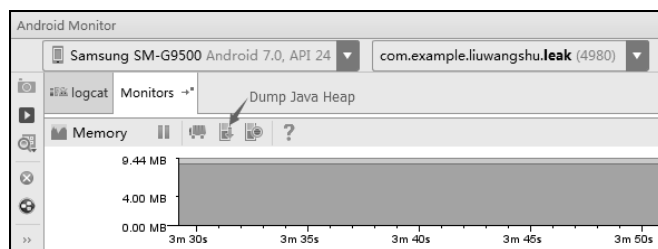


图 17-15 Memory Monitor 生成 hprof 文件

Memory Monitor 生成的 hprof 文件也不是标准的, AS 提供了便捷的转换方式: Memory Monitor 生成的 hprof 文件都会显示在 AS 左侧的 Captures 标签中, 在 Captures 标签中选择要转换的 hprof 文件, 并单击鼠标右键, 在弹出的菜单中选择 Export to standard.hprof 选项, 即可导出标准的 hprof 文件, 如图 17-16 所示。

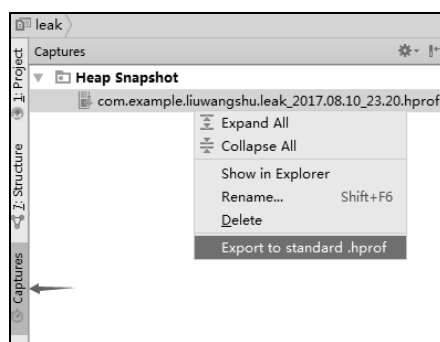


图 17-16 导出标准的 hprof 文件

17.5.2 MAT分析hprof文件

用 MAT 打开标准的 hprof 文件, 选择 Leak Suspects Report 选项, 这时 MAT 就会生成报告, 这个报告分为两个标签页, 一个是 Overview, 另一个是 Leak Suspects (内存泄漏猜想), 如图 17-17 所示。

在 Leak Suspects 中给出了 MAT 认为可能出现内存泄漏问题的地方, 图 17-17 共给出了 3 个内存泄漏猜想, 通过单击每个内存泄漏猜想的 Details 可以看到更深入的分析清理情况。如果内存泄漏不是特别明显, 通过 Leak Suspects 很难发现内存泄漏的位置。

打开 Overview 标签页, 首先看到的是一个饼状图, 它主要用来显示内存的消耗, 饼状图的彩色区域代表被分配的内存, 灰色区域则是空闲内存, 单击每个彩色区域可以看到这块区域的详细信息, 如图 17-18 所示。

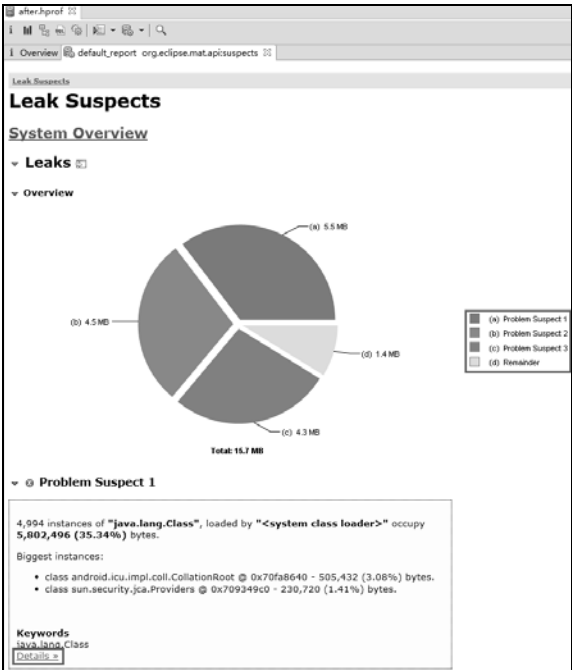


图 17-17 生成报告

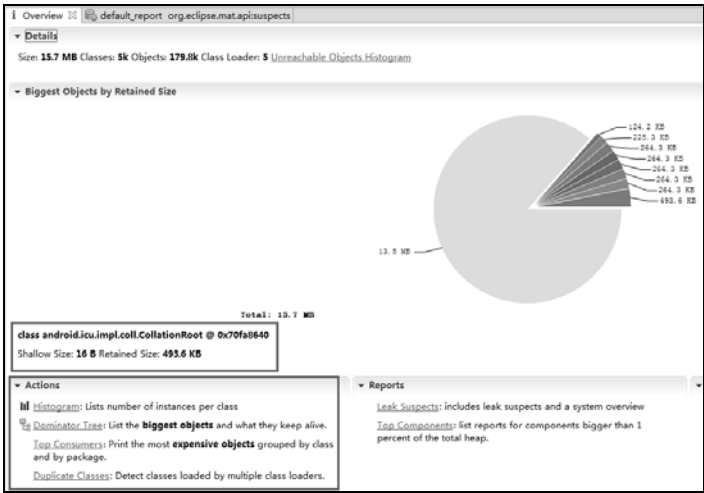


图 17-18 Overview 标签页

再往下看，Actions 一栏的下面列出了 MAT 提供的 4 种 Action，其中分析内存泄漏最常用的就是 Histogram 和 Dominator Tree。我们单击 Actions 中给出的链接或者在 MAT 工具栏中单击相应的选项就可以打开 Dominator Tree 和 Histogram，MAT 工具栏如图 17-19 所示。

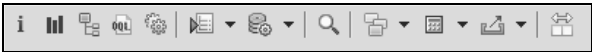


图 17-19 MAT 工具栏

其中左边第二个选项是 Histogram，第三个选项是 Dominator Tree，第四个选项是 OQL，下面分别对它们进行介绍。

17.5.2.1 Dominator Tree

Dominator Tree 意思为支配树，从名称就可以看出 Dominator Tree 更善于去分析对象的引用关系，如图 17-20 所示。

Class Name	Shallow Heap	Retained Heap	Percentage
<Regex>	<Numeric>	<Numeric>	<Numeric>
class android.support.v4.view.ViewPropertyAnimatorUpdateListener @ 0x12c61...	0	0	0.00%
class android.support.v7.app.ActionBar\$TabListener @ 0x12c63c00 System Cl...	0	0	0.00%
class android.support.v7.app.ActionBar\$OnMenuVisibilityListener @ 0x12c63d6...	0	0	0.00%
class android.support.v7.app.ActionBar\$OnNavigationListener @ 0x12c63e00 i...	0	0	0.00%
class android.support.v4.view.ViewCompat\$OnApplyWindowInsetsListe...	0	0	0.00%
class android.support.v4.view.NestedScrollingChild @ 0x12c6f680 System Clas...	0	0	0.00%
class com.sec.enterprise.knox.keystore.ITimeKeyStore @ 0x12c6f780 System C...	0	0	0.00%

图 17.20 Dominator Tree

- **Shallow Heap**：对象自身占用的内存大小，不包括它引用的对象。如果是数组类型的对象，它的大小由数组元素的类型和数组长度决定。如果是非数组类型的对象，它的大小由其成员变量的数量和类型决定。
- **Retained Heap**：一个对象的 Retained Set 包含对象所占内存的总大小。换句话说，Retained Heap 就是当前对象被 GC 后，从 Heap 上总共能释放掉的内存。

Retained Set 指的是这个对象本身和它持有引用的对象以及这些引用对象的 Retained Set 所占内存大小的总和，引用树官方的图解如图 17-21 所示。

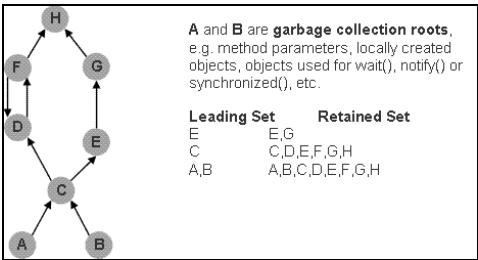


图 17-21 引用树官方的图解

从图 17-20 中可以看出 E 的 Retained Set 为 E 和 G。C 的 Retained Set 为 C、D、E、F、G、H。MAT 所定义的支配树就是从图 17-21 所示的引用树演化而来。在引用树当中，如果

一条到 Y 的路径必然会经过 X，称为 X 支配 Y。X 直接支配 Y 则指的是在所有支配 Y 的对象中，X 是 Y 最近的一个对象。支配树反映的就是这种直接支配关系，在支配树中，父节点直接支配子节点。如图 17-22 所示就是官方提供的一个从引用树到支配树的转换示意图。

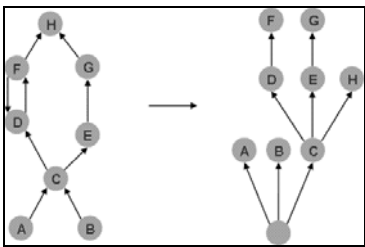


图 17-22 引用树到支配树的转换示意图

C 直接支配 D、E，因此 C 是 D、E 的父节点，这一点根据上面的阐述很容易得出结论。C 直接支配 H，这可能会有些疑问，能到达 H 主要有两条路径，而这两条路径 FD 和 GE 都不是必须要经过的节点，只有 C 满足了这一点，因此 C 直接支配 H，C 就是 H 的父节点。通过支配树，我们就可以很容易地分析一个对象的 Retained Set，比如 E 被回收，则会释放 E、G 的内存，而不会释放 H 的内存，因为 F 可能还引用着 H，只有 C 被回收，H 的内存才会被释放。这里对支配树进行了讲解，我们可以得出一个结论：通过 MAT 提供的 Dominator Tree，可以很清晰地得到一个对象的直接支配对象，如果直接支配对象中出现了不该有的对象，就说明发生了内存泄漏。在 Dominator Tree 的顶部 Regex 可以输入过滤条件（支持正则表达式），如果是查找 Activity 内存泄漏，可以在 Regex 中输入 Activity 的名称，比如这个例子可以输入 MainActivity，Dominator Tree 的信息如图 17-23 所示。

Class Name	Shallow Heap	Retained Heap	Percentage
«MainActivity»	«Numeric»	«Numeric»	«Numeric»
class com.example.liuwangshu.leak.MainActivity @ 0x12d17000 System Class	16	4,760	0.03%
com.example.liuwangshu.leak.MainActivity @ 0x12cd1340	280	1,120	0.01%
com.example.liuwangshu.leak.MainActivity @ 0x12cd1cd0	280	1,120	0.01%
com.example.liuwangshu.leak.MainActivity @ 0x12cd1ef0	280	1,120	0.01%
com.example.liuwangshu.leak.MainActivity @ 0x12cd230	280	1,120	0.01%
com.example.liuwangshu.leak.MainActivity @ 0x12cd450	280	1,120	0.01%
com.example.liuwangshu.leak.MainActivity @ 0x12cd670	280	1,120	0.01%
com.example.liuwangshu.leak.MainActivity @ 0x12cd890	280	1,120	0.01%
com.example.liuwangshu.leak.MainActivity @ 0x12cdab0	280	1,120	0.01%
com.example.liuwangshu.leak.MainActivity @ 0x12cdcd0	280	1,120	0.01%
com.example.liuwangshu.leak.MainActivity @ 0x12cded0	280	1,120	0.01%
com.example.liuwangshu.leak.MainActivity @ 0x12cf230	280	1,120	0.01%
class com.example.liuwangshu.leak.MainActivity\$LeakThread @ 0x12d26c00 \$y	16	920	0.01%
com.example.liuwangshu.leak.MainActivity\$LeakThread @ 0x12cb1e50 Thread	136	216	0.00%
com.example.liuwangshu.leak.MainActivity\$LeakThread @ 0x12cd2700 Thread	136	216	0.00%
com.example.liuwangshu.leak.MainActivity\$LeakThread @ 0x12cd9280 Thread	136	216	0.00%
com.example.liuwangshu.leak.MainActivity\$LeakThread @ 0x12cd9c10 Thread	136	216	0.00%
com.example.liuwangshu.leak.MainActivity\$LeakThread @ 0x12cd9d0 Thread	136	216	0.00%
com.example.liuwangshu.leak.MainActivity\$LeakThread @ 0x12d298b0 Thread	136	216	0.00%
com.example.liuwangshu.leak.MainActivity\$LeakThread @ 0x12d75160 Thread	136	216	0.00%
com.example.liuwangshu.leak.MainActivity\$LeakThread @ 0x12d75a60 Thread	136	216	0.00%
com.example.liuwangshu.leak.MainActivity\$LeakThread @ 0x12f16790 Thread	136	216	0.00%
com.example.liuwangshu.leak.MainActivity\$LeakThread @ 0x12f16f70 Thread	136	216	0.00%
com.example.liuwangshu.leak.MainActivity\$LeakThread @ 0x12f69820 Thread	136	216	0.00%
Σ Total: 24 entries (63,510 filtered)			

图 17-23 Dominator Tree 的信息

在 Dominator Tree 中列出了很多 MainActivity 实例, MainActivity 是不该有这么多个实例的, 基本可以断定发生了内存泄漏, 具体内存泄漏的原因, 可以查看 GC 引用链。在 MainActivity 选项上单击鼠标右键, 在弹出的菜单中选择 Path To GC Roots 选项, 如图 17-24 所示。

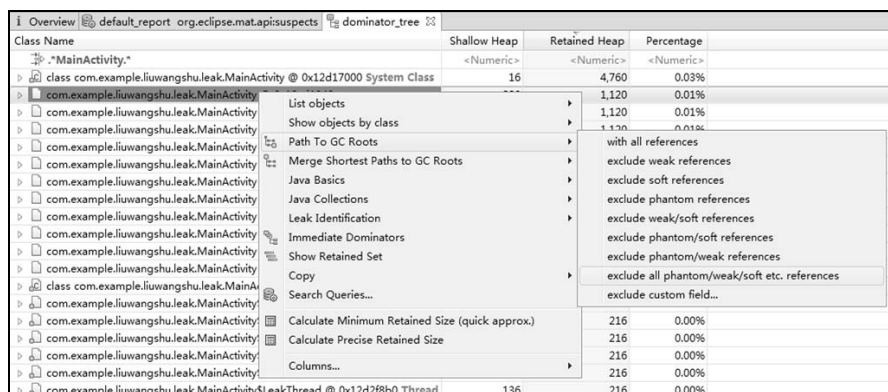


图 17-24 选择 Path To GC Roots 选项

Path To GC Roots 选项用来表示从对象到 GC Roots 的路径, 根据引用类型会有多种选项, 比如 with all references 就是包含所有的引用, 这里我们选择 exclude all phantom/weak/soft etc. references, 因为这个选项排除了虚引用、弱引用和软引用, 这些引用一般是可以被回收的, 这时 MAT 就会给出 MainActivity 的 GC 引用链, 如图 17-25 所示。

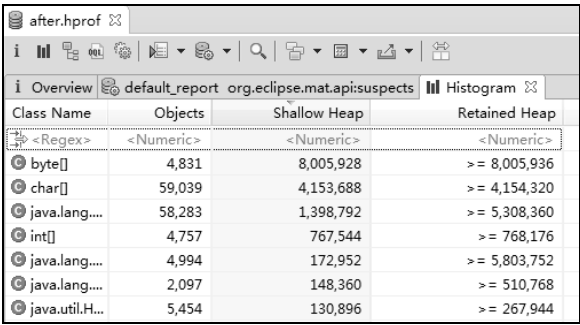
Class Name	Shallow Heap	Retained Heap
<Regex>	<Numeric>	<Numeric>
com.example.liuwangshu.leak.MainActivity @ 0x12cd1340	280	1,120
this\$0 com.example.liuwangshu.leak.MainActivity\$LeakThread @ 0x12cb1e50 Thread	136	216
mContext, mOnWindowDismissedCallback, mWindowControllerCallback com.android.	368	15,880
Σ Total: 2 entries		

图 17-25 MainActivity 的 GC 引用链

引用 MainActivity 的是 LeakThread, this\$0 的含义就是内部类自动保留的一个指向所在外部类的引用, 而这个外部类就是 MainActivity, 这将会导致 MainActivity 无法被 GC。

17.5.2.2 Histogram

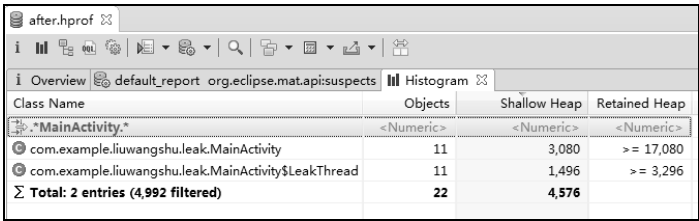
Histogram 与 Dominator Tree 不同的是, Dominator Tree 是从对象实例的角度进行分析, 注重引用关系分析, 而 Histogram 则从类的角度进行分析, 注重量的分析。Histogram 中的内容如图 17-26 所示。



Class Name	Objects	Shallow Heap	Retained Heap
<Regex>	<Numeric>	<Numeric>	<Numeric>
byte[]	4,831	8,005,928	>= 8,005,936
char[]	59,039	4,153,688	>= 4,154,320
java.lang....	58,283	1,398,792	>= 5,308,360
int[]	4,757	767,544	>= 768,176
java.lang....	4,994	172,952	>= 5,803,752
java.lang....	2,097	148,360	>= 510,768
java.util.H...	5,454	130,896	>= 267,944

图 17-26 Histogram 中的内容

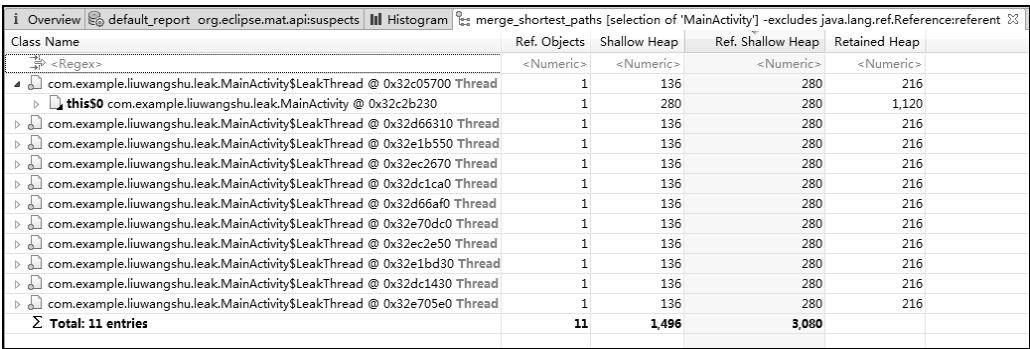
在 Histogram 中共包含 4 列数据,关于 Shallow Heap 和 Retained Heap 的含义在 17.5.2.1 节已经介绍过了,剩余的 Class Name 代表类名, Objects 代表对象实例的个数。在 Histogram 的顶部 Regex 同样可以输入过滤条件,这里同样输入 MainActivity,效果如图 17-27 所示。



Class Name	Objects	Shallow Heap	Retained Heap
.*MainActivity.*	<Numeric>	<Numeric>	<Numeric>
com.example.liuwangshu.leak.MainActivity	11	3,080	>= 17,080
com.example.liuwangshu.leak.MainActivity\$LeakThread	11	1,496	>= 3,296
Σ Total: 2 entries (4,992 filtered)	22	4,576	

图 17-27 顶部 Regex 输入 MainActivity

MainActivity 和 LeakThread 实例各为 11 个,基本上可以断定发生了内存泄漏。具体内存泄漏的原因,同样可以查看 GC 引用链。在 MainActivity 选项上单击鼠标右键,在弹出的菜单中选择 Merge Shortest Paths to GC roots,并在选项中选择 exclude all phantom/weak/soft etc. references, Path To GC Roots 的信息如图 17-28 所示。



Class Name	Ref. Objects	Shallow Heap	Ref. Shallow Heap	Retained Heap
<Regex>	<Numeric>	<Numeric>	<Numeric>	<Numeric>
com.example.liuwangshu.leak.MainActivity\$LeakThread @ 0x32c05700 Thread	1	136	280	216
this\$0 com.example.liuwangshu.leak.MainActivity @ 0x32c2b230	1	280	280	1,120
com.example.liuwangshu.leak.MainActivity\$LeakThread @ 0x32d66310 Thread	1	136	280	216
com.example.liuwangshu.leak.MainActivity\$LeakThread @ 0x32e1b550 Thread	1	136	280	216
com.example.liuwangshu.leak.MainActivity\$LeakThread @ 0x32ec2670 Thread	1	136	280	216
com.example.liuwangshu.leak.MainActivity\$LeakThread @ 0x32dc1ca0 Thread	1	136	280	216
com.example.liuwangshu.leak.MainActivity\$LeakThread @ 0x32d66af0 Thread	1	136	280	216
com.example.liuwangshu.leak.MainActivity\$LeakThread @ 0x32e70dc0 Thread	1	136	280	216
com.example.liuwangshu.leak.MainActivity\$LeakThread @ 0x32ec2e50 Thread	1	136	280	216
com.example.liuwangshu.leak.MainActivity\$LeakThread @ 0x32e1bd30 Thread	1	136	280	216
com.example.liuwangshu.leak.MainActivity\$LeakThread @ 0x32dc1430 Thread	1	136	280	216
com.example.liuwangshu.leak.MainActivity\$LeakThread @ 0x32e705e0 Thread	1	136	280	216
Σ Total: 11 entries	11	1,496	3,080	

图 17-28 Path To GC Roots 的信息

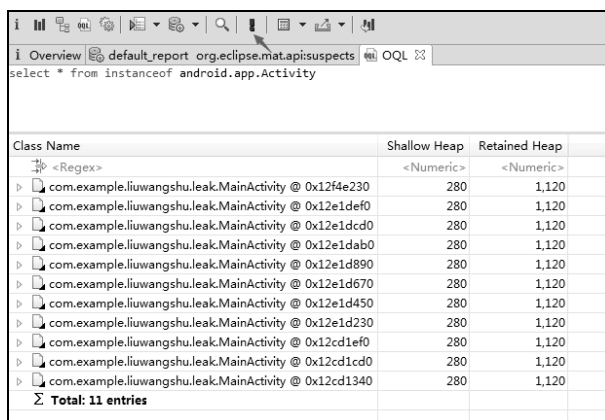
Histogram 是从类的角度进行分析，而 Path To GC Roots 是用来分析单个对象的，因此在 Histogram 中无法使用 Path To GC Roots 查询，可以使用 Merge Shortest Paths to GC roots 查询，它表示从 GC roots 到一个或一组对象的公共路径。得出的结果和 17.5.2.1 节是相同的，引用 MainActivity 的是 LeakThread，这导致了 MainActivity 无法被 GC。

17.5.2.3 OQL

OQL 全称为 Object Query Language，类似于 SQL 语句的查询语言，能够用来查询当前内存中满足指定条件的所有对象。它的查询语句的基本格式如下：

```
SELECT * FROM [ INSTANCEOF ] <class_name> [ WHERE <filter-expression> ]
```

当输入 `select * from instanceof android.app.Activity` 并按下 F5 键时（或者单击工具栏中的红色叹号），将当前内存中所有 Activity 都显示出来，如图 17-29 所示。



Class Name	Shallow Heap	Retained Heap
<Regex>	<Numeric>	<Numeric>
com.example.liuwangshu.leak.MainActivity @ 0x12f4e230	280	1,120
com.example.liuwangshu.leak.MainActivity @ 0x12e1def0	280	1,120
com.example.liuwangshu.leak.MainActivity @ 0x12e1dcd0	280	1,120
com.example.liuwangshu.leak.MainActivity @ 0x12e1dab0	280	1,120
com.example.liuwangshu.leak.MainActivity @ 0x12e1d890	280	1,120
com.example.liuwangshu.leak.MainActivity @ 0x12e1d670	280	1,120
com.example.liuwangshu.leak.MainActivity @ 0x12e1d450	280	1,120
com.example.liuwangshu.leak.MainActivity @ 0x12e1d230	280	1,120
com.example.liuwangshu.leak.MainActivity @ 0x12cd1ef0	280	1,120
com.example.liuwangshu.leak.MainActivity @ 0x12cd1cd0	280	1,120
com.example.liuwangshu.leak.MainActivity @ 0x12cd1340	280	1,120
Σ Total: 11 entries		

图 17-29 OQL 查询

如果 Activity 比较多，或者你想查找具体的类，可以直接输入具体类的完整名称：

```
select * from com.example.liuwangshu.leak.MainActivity
```

通过查看 GC 引用链也可以找到内存泄漏的原因。关于 OQL 语句有很多用法，这里就不过多介绍了，具体可以查看官方文档：<http://help.eclipse.org/luna/index.jsp?topic=/org.eclipse.mat.ui.help/reference/oqlsyntax.html>。

17.5.2.4 对比 hprof 文件

因为这一节的例子很简单，可以通过上面的方法可以找到内存泄漏的原因，但是复杂的情况就需要通过对比 hprof 文件来进行分析了。使用步骤如下：

- (1) 操作应用，生成第一个 hprof 文件。
- (2) 进行一段时间操作，再生成第二个 hprof 文件。
- (3) 用 MAT 打开这两个 hprof 文件。
- (4) 将第一个和第二个 hprof 文件的 Dominator Tree 或者 Histogram 添加到 Compare Basket 中，如图 17-30 所示。

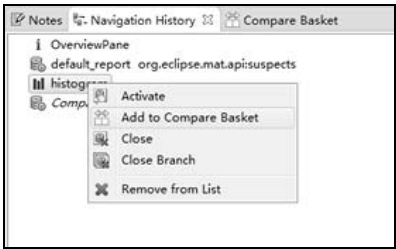


图 17-30 添加到 Compare Basket 的操作

- (5) 在 Compare Basket 中单击红色叹号按钮生成 Compared Tables, Compared Tables 如图 17-31 所示。

Class Name	Objects #0	Objects #1	Shallow Heap #0	Shallow Heap #1
<Regex>	<Numeric>	<Numeric>	<Numeric>	<Numeric>
android.R.styleable	0	0	0	0
android.accounts.Account	0	0	0	0
android.accounts.Account\$1	1	1	8	8
android.accounts.AccountManager	0	0	0	0
android.accounts.AccountManager\$1	0	0	0	0
android.accounts.AccountManager\$11	0	0	0	0
android.accounts.AccountManager\$AmsTask	0	0	0	0
android.accounts.AccountManager\$AmsTask\$1	0	0	0	0
android.accounts.AccountManager\$AmsTask\$Respo...	0	0	0	0

图 17-31 Compared Tables

- (6) 在 Compared Tables 也有顶部 Regex，输入 MainActivity 进行筛选，如图 17-32 所示。

Class Name	Objects	Shallow Heap
MainActivity.	<Numeric>	<Numeric>
com.example.liuwangshu.leak....	+6	+1,488
com.example.liuwangshu.leak....	+6	+816
Σ Total: 2 entries (4,958 filtered)	+12	+2,304

图 17-32 在 Regex 中筛选

MainActivity 在这一过程中增加了 6 个，MainActivity 的实例是不应该增加的，这说明发生了内存泄漏，可以通过查看 GC 引用链来找到内存泄漏的具体原因。除了上面的对比

方法，Histogram 还可以通过工具栏的对比按钮来进行对比，对比按钮见图 17-33 中的箭头指向。

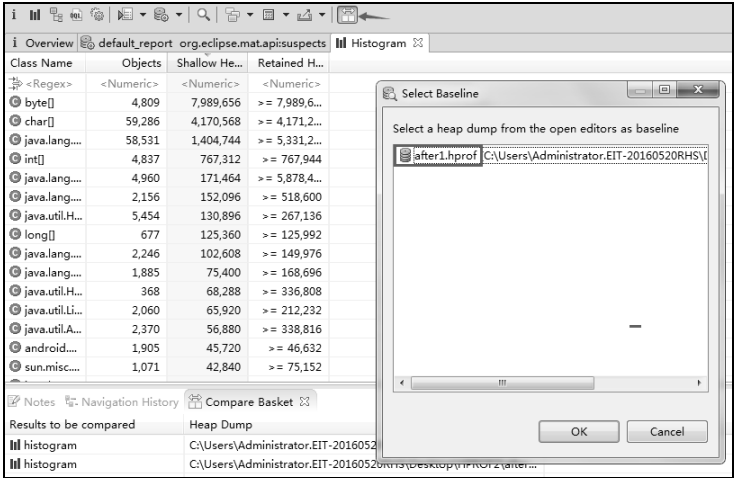


图 17-33 对比按钮

(7) 生成的结果和 Compared Tables 类似，我们输入 MainActivity 进行筛选，如图 17-34 所示。

Overview	default_report	org.eclipse.mat.api:suspects	Histogram
Class Name	Objects	Shallow Heap	
<Regex>	<Numeric>	<Numeric>	
byte[]	4,809	7,989,656	>= 7,989,6...
char[]	59,286	4,170,568	>= 4,171,2...
java.lang....	58,531	1,404,744	>= 5,331.2...
int[]	4,837	767,312	>= 767,944
java.lang....	4,960	171,464	>= 5,878,4...
java.lang....	2,156	152,096	>= 518,600
java.util.H...	5,454	130,896	>= 267,136
long[]	677	125,360	>= 125,992
java.lang....	2,246	102,608	>= 149,976
java.lang....	1,885	75,400	>= 168,696
java.util.H...	368	68,288	>= 336,808
java.util.Li...	2,060	65,920	>= 212,232
java.util.A...	2,370	56,880	>= 338,816
android....	1,905	45,720	>= 46,632
sun.misc....	1,071	42,840	>= 75,152

Results to be compared	Heap Dump
histogram	C:\Users\Administrator.EIT-20160520RHSU...
histogram	C:\Users\Administrator.EIT-20160520RHSU... (desktop\p\hprof\after1...)

图 17-34 输入 MainActivity 进行筛选

可以看到第二个 hprof 文件比第一个 hprof 文件多了 6 个 MainActivity 实例。

MAT 还有很多功能，这里也只介绍了常用的功能，其他的功能就需要读者在使用过程中去发现并积累。

17.6 LeakCanary

如果使用MAT来分析内存问题，会有一些难度，并且效率也不是很高，对于一个内存泄漏问题，可能要进行多次排查和对比。为了能够迅速地发现内存泄漏，Square公司基于MAT开源了LeakCanary，地址为<https://github.com/square/leakcanary>。

17.6.1 使用LeakCanary

首先配置 build.gradle:

```
dependencies {
    debugCompile 'com.squareup.leakcanary:leakcanary-android:1.5.2'
    releaseCompile 'com.squareup.leakcanary:leakcanary-android-no-op:1.5.2'
}
```

接下来在 Application 中加入如下代码:

```
public class LeakApplication extends Application {
    @Override public void onCreate() {
        super.onCreate();
        if (LeakCanary.isInAnalyzerProcess(this)) { //1
            return;
        }
        LeakCanary.install(this);
    }
}
```

注释 1 处的代码用来进行过滤操作,如果当前的进程是用来给 LeakCanary 进行堆分析的则返回,否则会执行 LeakCanary 的 install 方法。这样我们就可以使用 LeakCanary 了,如果检测到某个 Activity 有内存泄漏,LeakCanary 就会给出提示。

17.6.2 LeakCanary应用举例

17.6.1 节的例子代码只能够检测 Activity 的内存泄漏,如果还需要检测其他类的内存泄漏,我们就需要使用 RefWatcher 来进行监控。改写 Application, 如下所示:

```
public class LeakApplication extends Application {
    private RefWatcher refWatcher;
    @Override
    public void onCreate() {
        super.onCreate();
        refWatcher= setupLeakCanary();
    }
    private RefWatcher setupLeakCanary() {
        if (LeakCanary.isInAnalyzerProcess(this)) {
            return RefWatcher.DISABLED;
        }
        return LeakCanary.install(this);
    }
}
```

```

    public static RefWatcher getRefWatcher(Context context) {
        LeakApplication leakApplication = (LeakApplication) context.getApplication
            Context();
        return leakApplication.refWatcher;
    }
}

```

install 方法会返回 RefWatcher 用来监控对象, LeakApplication 中还要提供 getRefWatcher 静态方法来返回全局 RefWatcher。最后为了举例, 我们在一段存在内存泄漏的代码中引入 LeakCanary 监控, 如下所示:

```

public class MainActivity extends AppCompatActivity {
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        LeakThread leakThread = new LeakThread();
        leakThread.start();
    }
    class LeakThread extends Thread {
        @Override
        public void run() {
            try {
                Thread.sleep(6 * 60 * 1000);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
    @Override
    protected void onDestroy() {
        super.onDestroy();
        RefWatcher refWatcher = LeakApplication.getRefWatcher(this);//1
        refWatcher.watch(this);
    }
}

```

MainActivity 存在内存泄漏, 原因就是非静态内部类 LeakThread 持有外部类 MainActivity 的引用, 在 LeakThread 中做了耗时操作, 导致 MainActivity 无法被释放。在注释 1 处得到 RefWatcher, 并调用它的 watch 方法, watch 方法的参数就是要监控的对象。当然, 在这个例子中 onDestroy 方法是多余的, 因为 LeakCanary 在调用 install 方法时会启动一个 ActivityRefWatcher 类, 它用于自动监控 Activity 执行 onDestroy 方法之后是否发生内存泄漏。这里只是为了方便举例, 如果想要监控 Fragment, 在 Fragment 中添加如上的 onDestroy

方法是有用的。运行程序，这时会在界面上生成一个名为 Leaks 的应用图标。接下来不断地切换横竖屏，这时会闪出一个提示框，提示内容为“Dumping memory app will freeze.Brrrr.”。再稍等片刻，内存泄漏信息就会通过 Notification 展示出来，比如三星 S8 的通知栏如图 17-35 所示。



图 17-35 三星 S8 的通知栏

在 Notification 中提示 MainActivity 发生了内存泄漏，泄漏的内存为 787B。点击 Notification 就可以进入内存泄漏详情页，除此之外也可以通过 Leaks 应用的列表界面进入。内存泄漏详情页如图 17-36 所示。

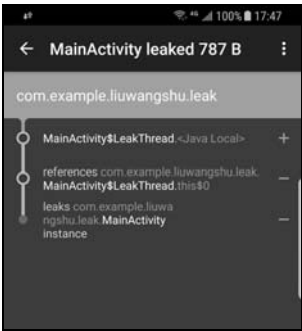


图 17-36 内存泄漏详情页

点击加号就可以查看具体类所在的包名称。整个详情就是一个引用链：MainActivity 的内部类 LeakThread 引用了 LeakThread 的 this\$0，this\$0 的含义就是内部类自动保留的一个指向所在外部类的引用，而这个外部类就是详情页最后一行所给出的 MainActivity 的实例，这将导致 MainActivity 无法被 GC，从而产生内存泄漏。

除此之外，我们还可以将 heap dump (hprof 文件) 和 info 信息分享出去，如图 17-37 所示。

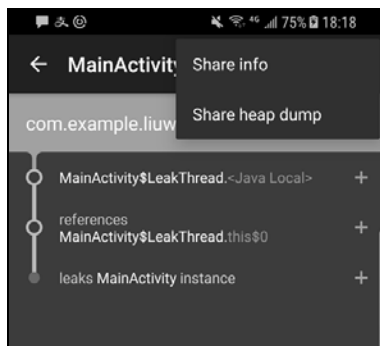


图 17-37 在内存泄漏详情页中分享信息

需要注意的是分享出去的 hprof 文件并不是标准的 hprof 文件，还需要将它转换为标准的 hprof 文件，这样才会被 MAT 识别从而进行分析。最后说一下解决方法就是将 LeakThread 改为静态内部类，代码如下所示：

```
public class MainActivity extends AppCompatActivity {
    ...
    static class LeakThread extends Thread {
        @Override
        public void run() {
            try {
                Thread.sleep(6 * 60 * 1000);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
    ...
}
```

再次运行程序 LeakThread 就不会给出内存泄漏的提示了。

17.7 本章小结

本章的知识点稍微有点多，首先介绍了在应用开发中如何避免可控的内存泄漏，接下来介绍了常用的内存分析工具：Memory Monitor、Allocation Tracker 和 Heap Dump，这些工具可以对内存进行分析，找到疑似发生内存泄漏的位置，如果想要深入地对内存进行分析并确定内存泄漏还需要学习 MAT 的使用方法，当然 MAT 的使用会有一些难度，并且效率也不是很高，因此最后介绍了 LeakCanary 的使用方法。这些工具的使用都是基本的入门方法，想要进阶的话还需要读者在项目中更多地进行实践。

反侵权盗版声明

电子工业出版社依法对本作品享有专有出版权。任何未经权利人书面许可，复制、销售或通过信息网络传播本作品的行为；歪曲、篡改、剽窃本作品的行为，均违反《中华人民共和国著作权法》，其行为人应承担相应的民事责任和行政责任，构成犯罪的，将被依法追究刑事责任。

为了维护市场秩序，保护权利人的合法权益，我社将依法查处和打击侵权盗版的单位和个人。欢迎社会各界人士积极举报侵权盗版行为，本社将奖励举报有功人员，并保证举报人的信息不被泄露。

举报电话：(010)88254396；(010)88258888

传 真：(010)88254397

E-mail: dbqq@phei.com.cn

通信地址：北京市万寿路 173 信箱 电子工业出版社总编办公室

